2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

PARALLEL-PROCESSOR-BASED GAUSSIAN BEAM TRACER FOR USE IN OCEAN ACOUSTIC TOMOGRAPHY

by

Roderick Spencer Scott

June 1990

Thesis Advisor:	J.H. Miller
Co-Advisor:	C.S. Chiu
Co-Advisor:	C. Yang

Unclassified

Security Classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified	1b Restrictive Markings
2a Security Classification Authority	3 Distribution Availability of Report
2b Declassification/Downgrading Schedule	Approved for public release; distribution is unlimited
4 Performing Organization Report Number(s)	5 Monitoring Organization Report Number(s)
6a Name of Performing Organization Naval Postgraduate School; 6b Office Symbol *(If Applicable)* 62Mr	7a Name of Monitoring Organization Naval Postgraduate School
6c Address *(city, state, and ZIP code)* Monterey, CA 93943-5000	7b Address *(city, state, and ZIP code)* Monterey, CA 93943-5000
8a Name of Funding/Sponsoring Organization; 8b Office Symbol *(If Applicable)*	9 Procurement Instrument Identification Number
8c Address *(city, state, and ZIP code)*	10 Source of Funding Numbers: Program Element Number / Project No / Task No / Work Unit Accession No

11 Title *(Include Security Classification)* PARALLEL-PROCESSOR-BASED GAUSSIAN BEAM TRACER FOR USE IN OCEAN ACOUSTIC TOMOGRAPHY

12 Personal Author(s) Scott, Roderick, S.

13a Type of Report	13b Time Covered	14 Date of Report *(year, month,day)*	15 Page Count
Master's Thesis	From To	June 1990	137

16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 Cosati Codes			18 Subject Terms *(continue on reverse if necessary and identify by block number)*
Field	Group	Subgroup	Acoustic Tomography, Ray Tracing, Parallel Processing, Gaussian Beams, Transputers

19 Abstract *(continue on reverse if necessary and identify by block number)*

This thesis presents a parallel-processor-based acoustic ray tracing algorithm for use in predicting multipath arrival times and amplitudes, for use in ocean acoustic tomography experiments. The Runge-Kutta-Fehlberg numerical integration method was chosen, out of three other methods, to numerically solve the ray equations. Cubic splines were used to interpolate the sound speed profile and bottom bathymetry data. The method of Gaussian beam tracing was used to compute the multipath amplitudes at a given receiver location. The ray tracing algorithm is written in C, and is designed to run using a Macintosh II-based host application and a transputer based, parallel processing workfarm.

20 Distribution/Availability of Abstract [✓ unclassified/unlimited] same as report DTIC users	21 Abstract Security Classification Unclassified
22a Name of Responsible Individual James H. Miller	22b Telephone *(Include Area code)* (408) 646-2384; 22c Office Symbol 62Mr

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted security classification of this page

All other editions are obsolete Unclassified

Parallel-Processor-Based Gaussian Beam Tracer For Use in Ocean Acoustic Tomography

by

Roderick Spencer Scott
Captain, Canadian Forces
B.Eng., Royal Military College of Canada, 1984

Submitted in partial fullfillment of the
requirements for the degrees of

MASTER OF SCIENCE IN ENGINEERING ACOUSTICS

and

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1990

Author: [illegible]
Roderick Spencer Scott

Approved by: [illegible]
James H. Miller, Thesis Advisor

[illegible]
Ching-Sang Chiu, Thesis Co-Advisor

[illegible]
Chyan Yang, Thesis Co-Advisor

[illegible]
Anthony A. Atchley, Chairman,
Engineering Acoustics Academic Committee

[illegible]
John Powers, Chairman, Department of Electrical
and Computer Engineering

ABSTRACT

This thesis presents a parallel-processor-based acoustic ray tracing algorithm for use in predicting multipath arrival times and amplitudes, for use in ocean acoustic tomography experiments. The Runge-Kutta-Fehlberg numerical integration method was chosen, out of three other methods, to numerically solve the ray equations. Cubic splines were used to interpolate the sound speed profile and bottom bathymetry data. The method of Gaussian beam tracing was used to compute the multipath amplitudes at a given receiver location. The ray tracing algorithm is written in C, and is designed to run using a Macintosh II-based host application and a transputer-based, parallel processing workfarm.

THESIS DISCLAIMER

The reader is cautioned that computer programs developed for this research may not have been exercised for all cases of interest. While every effort has been made within the time available to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempting to cite each individual occurence of a trademark, all registered trademarks appearing in this thesis are listed below the firm holding the trademark:

Apple Computer Corporation, Cupertino, CA:
- Macintosh II
- MPW

INMOS Limited, Bristol, United Kingdom:
- Transputer
- Occam
- IMS T800

Levco, Inc., San Diego, CA:
- TransLink
- Link II

Symantec Corporation, Cupertino, CA:
- Think C

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to thank Dr. James H. Miller, my primary thesis advisor for his encouragement and support - without him, my thesis work would not have been so enjoyable. Also I would like to thank my co-advisors, Dr. Ching-Sang Chiu and Dr. Chyan Yang, for their expert advice, support and friendship.

Finally, I dedicate this thesis to my wife Claire and my children, Christine and Eric, who gave up so much and asked for so little in return. Without them it would not have been worthwhile - or even possible.

I. INTRODUCTION

A. OCEAN ACOUSTIC TOMOGRAPHY

"Ocean acoustic tomography is a technique for observing the dynamic behaviour of ocean processes by measuring the changes in travel time of acoustic signals transmitted over a number of ocean paths." (Spindel, 1986, pp. 7-13) Ocean acoustic tomography has been concerned with measuring the mesoscale fluctuations and features of the ocean which are characterized by dimensions in the hundreds of kilometers and time scales on the order of months.

The term tomography is derived from the Greek word "tomo" which means "cut" or "slice". Ocean acoustic tomography was originally proposed by Walter Munk and Carl Wunsch and methods for inverting observed data to obtain sound speed fluctuations were presented in Munk (1979). The speed at which sound travels through the ocean is a function of many factors including temperature, salinity and pressure. In addition, travel times can be affected by currents. By looking at a "slice" of the ocean between several points (moorings) using acoustic transmissions, the travel time information obtained is used to estimate these fluctuations in ocean variables. This is done using inverse techniques in a fashion similar to that done in Computer Assisted Tomography (CAT) where X-rays are used to yield a two-dimensional view of the interior of the human anatomy.

Measurements in the ocean can be made using a number of acoustic sources (S) and receivers (R) as depicted in Figure 1.1. Whereas spot measurements would yield $S+R$ pieces of information and would be contaminated by small scale fluctuations, the tomography approach is multiplicative and yields $S \times R$ pieces of information. Also measurements of travel time tend to be spatially integrating which effectively smoothes out the small scale fluctuations. (Munk, 1979, p. 124)

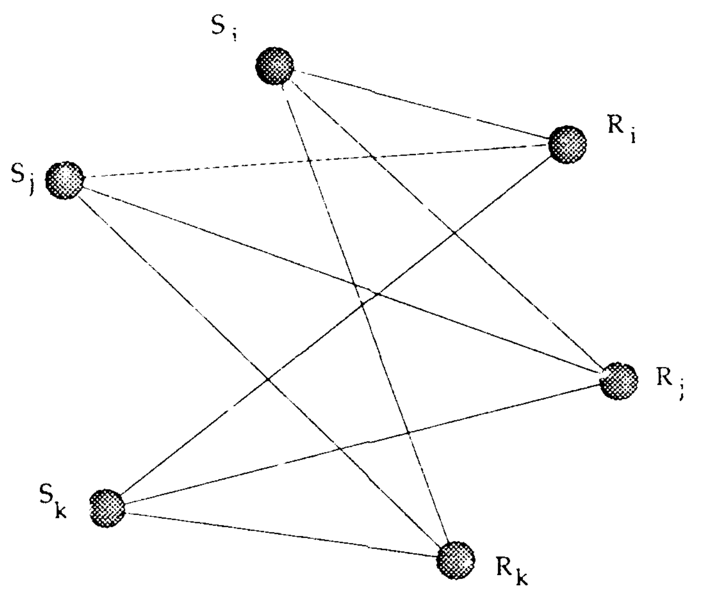

Figure 1.1 **Example Source and Receiver Array**

Of importance in ocean acoustic tomography is the ability to estimate the possible acoustic paths (multipaths) between a source and receiver and the information such as travel time and energy associated with them. One such method is ray tracing which uses geometrical principles to determine the paths taken by the acoustic energy. Although ray tracing is based on certain limiting assumptions, it provides an intuitive and visual means of computing the acoustic field and the paths followed by the acoustic wave fronts.

Recently, it has been suggested that tomography could monitor the circulation of Monterey Bay, California (Miller et al., 1989). The application of traditional ray tracing programs has been hampered by their inability to model propagation through the extreme bathymetry of the Monterey Bay submarine canyon. The traditional programs have relied on the determination of eigenrays (rays which connect source and receiver) by shooting methods. Rays are traced from the source at specified angles and travel past the receiver range. Eigenrays can be estimated by interpolating between rays which bracket the receiver depth. This is a very computer-intensive operation for typical deep ocean tomography scenarios. The extreme bathymetry of Monterey Bay makes it even more so. This thesis attempts to make this modelli. ̃ problem more tractable through the use of Gaussian beam tracing and with a parallel-processor-based workfarm system running in a Macintosh II desktop computer.

B. THESIS ORGANIZATION

This thesis is organized into eight chapters. Chapter II discusses the basic theory of geometrical acoustics, or ray tracing. Some basic equations are derived and common methods for solving these equations are given. A method of using cubic splines to interpolate sound speed profile data is presented. Chapter III presents four numerical integration techniques that were considered for this thesis to solve the basic ray equation. A brief discussion of numerical integration and each of the methods is given. The integration methods were tested using a simple ray tracing problem to see which method was most suitable. The re^sults of this test are presented.

Chapter IV gives a discussion of a relatively new method for computing acoustic fields in the ocean – Gaussian beam tracing. This method associates with each ray path, a beam with a Gaussian distribution, and can be used to scale other acoustic quantities s''ch as intensity and pressure. The Gaussian beam method is also free of certain artifacts which are present in conventional ray tracing techniques, such as infinite energy at caustics and perfect shadow zones.

Some basic concepts about parallel processing are presented in Chapter V. The architecture of the T800 transputer and a discussion about the arallel processor workfarm is also presented. Chapter VI presents some implementation aspects of this thesis. Chapter VII presents some results obtained using the ray tracing and gaussian beam algorithms and results from the parallel processing optimization. Chapter VIII presents conclusions and recommendations for further work in this area.

II. RAY TRACING

A. RAY THEORY

The linearized, lossless wave equation for the propagation of sound in fluids is given by

$$\nabla^2 p = \frac{1}{c^2}\frac{\partial^2 p}{\partial t^2}, \tag{2.1}$$

where c is the phase speed of acoustic waves in the fluid and p is the acoustic pressure. Both c and p are functions of position (i.e., c = c(x,y,z), p = p(x,y,z)). It is often useful to think of the propagation of sound in terms of rays instead of plane waves. Rays can be defined as lines which are perpendicular everywhere to surfaces of constant phase (Kinsler, 1982, p. 117). In many cases rays are easier to work with than waves; however rays are approximations and are only valid under certain conditions. One possible solution to the wave equation which leads to the ray approximation is given by

$$p(x,y,z,t) = A(x,y,z)\, e^{i\omega[t - \Gamma(x,y,z)/c_0]}, \tag{2.2}$$

where A is the pressure amplitude, Γ is a function with units of length and c_0 is a constant value of phase speed. Surfaces of constant phase are defined by values of (x,y,z) such that Γ is constant. The quantity $\nabla\Gamma$ is therefore perpendicular to these surfaces of constant phase. Substituting Equation (2.2) into the wave equation yields

$$\frac{\nabla^2 A}{A} - \left\{\frac{\omega}{c_o}\right\}^2 \nabla\Gamma\bullet\nabla\Gamma + \left\{\frac{\omega}{c}\right\}^2 - i\frac{\omega}{c_o}\left\{2\,\frac{\nabla^2 A}{A}\bullet\nabla\Gamma + \nabla^2\Gamma\right\} = 0. \tag{2.3}$$

If A and $\nabla\Gamma$ vary slowly such that

$$\left|\frac{\nabla^2 A}{A}\right| \ll (\omega/c_o)^2\,,\ \ |\nabla^2\Gamma| \ll \omega/c\,,\ \ \left|\frac{\nabla A}{A}\bullet\nabla\Gamma\right| \ll \omega/c\,, \qquad (2.4)$$

then Equation (2.3) simplifies to the Eikonal equation

$$|\nabla\Gamma|^2 = n^2, \qquad (2.5)$$

where

$$n = n(x,y,z) = \frac{c_o}{c(x,y,z)} \qquad (2.6)$$

is the index of refraction. The conditions given in Equation (2.4) can be met if the amplitude of the wave and the speed of sound do "...not change appreciably in distances comparable to a wavelength." (Kinsler, 1982, pp. 117-118)

Solution of Equation (2.5) yields the ray path trajectories that the acoustic energy follows. The behaviour of $\nabla\Gamma$ is given by

$$\frac{d}{ds}(\nabla\Gamma) = \nabla n, \qquad (2.7)$$

which, if the sound speed is a function of depth only (i.e., $c = c(z)$), leads to a form of Snell's law

$$\frac{\cos\theta}{c(z)} = \text{constant}, \qquad (2.8)$$

where θ is the angle of the ray path measured from the horizontal at a particular location along the ray path. (Kinsler, 1982, pp. 118-120)

B. METHODS FOR SOLVING THE RAY PATH

1. Ray Equations

By the repeated application of Snell's law in a horizontally stratified medium, the ray path can be determined by

$$\frac{\cos\theta_1}{c_1} = \frac{\cos\theta_2}{c_2} = \cdots \frac{\cos\theta_n}{c_n}, \tag{2.9}$$

where the subscripts 1,2, ... n denote the successive layers of the medium (Clay, 1977, p. 83). Again, the speed of sound is assumed to vary only with depth. An element of the ray path at a point P is shown in Figure 2.1. The quantities ds, dz and dx represent the differential ray path length, vertical displacement and horizontal displacement respectively.

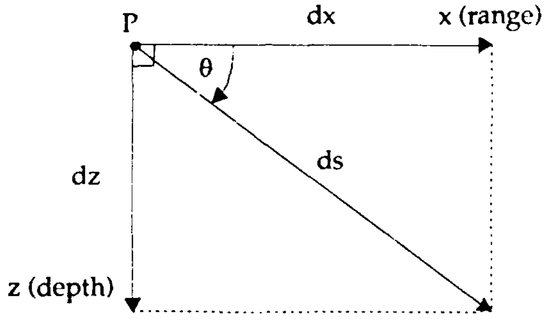

Figure 2.1 **Eiement of Ray Path at Point P**

From Figure 2.1 we have

$$ds^2 = dz^2 + dx^2. \tag{2.10}$$

Dividing both sides by dx and solving for the differential change in depth (dz) with respect to the differential change in range (dx) yields

$$\left(\frac{ds}{dx}\right)^2 = \left(\frac{dz}{dx}\right)^2 + 1, \tag{2.11}$$

or

$$\frac{dz}{dx} = \pm\sqrt{\left(\frac{ds}{dx}\right)^2 - 1}. \tag{2.12}$$

Also from Figure 2.2 we have the relation

$$\frac{ds}{dx} = \frac{1}{\cos\theta}. \tag{2.13}$$

From Snell's law we have

$$\frac{\cos\theta_o}{c_o} = \frac{\cos\theta}{c(z)} = a, \tag{2.14}$$

so that

$$\cos\theta = a\,c(z), \tag{2.15}$$

where a is the Snell's law constant of the ray (Clay, 1977, p. 84). Substituting this result into Equation (2.12) and taking the positive branch of the square root yields

$$\frac{dz}{dx} = \sqrt{\left(\frac{1}{a\,c(z)}\right)^2 - 1}, \tag{2.16}$$

the differential change in depth with respect to a differential change in range. The differential travel time along the ray segment is given by (Clay, 1977, p. 84)

$$\frac{dt}{ds} = \frac{1}{c(z)}, \tag{2.17}$$

or

$$dt = \frac{dx}{c(z)\cos\theta}. \tag{2.18}$$

These relations can also be expressed as integrals along the ray from an initial range x_i to a final range x_f

$$z_f - z_i = \int_{x_i}^{x_f} \frac{dx}{\sqrt{\left(\frac{1}{a\,c(z)}\right)^2 - 1}} \tag{2.19}$$

$$t_f - t_i = \int_{x_i}^{x_f} \frac{dx}{c(z)\cos\theta}. \tag{2.20}$$

Although the speed of sound is a continuous function of depth, a common approximation made in some ray tracing implementations is to break the sound speed profile into linear segments, or layers as shown in Figure 2.2. This is natural since the sound speed is usually only known (i.e., measured) at discrete depth points.

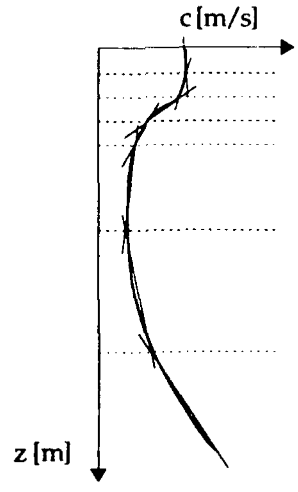

Figure 2.2 **Linear Approximation of Sound Speed Profile**

Note that the thicker curved line in Figure 2.2 represents the actual sound speed profile; the thin straight lines represent the linear approximations and the dashed lines represent the layers.

Once this linear approximation is made to the sound speed profile, the ray path within a particular layer can be evaluated analytically. The result is that the ray path follows the arc of a circle whose radius is given by

$$\text{radius} = \frac{1}{ag}, \tag{2.21}$$

where a is the Snell' s law constant defined in Equation (2.14) and g is the constant sound speed gradient (dc/dz) within the layer (Clay, 1977, p. 87). Although this method is simplistic and suitable for hand calculation, it can yield unsatisfactory results due to discontinuities in dc/dz at the interfaces (Pederson, 1961, pp. 465-474).

2. Sound Speed Profile Interpolation

To avoid the problems associated with using linearly segmented sound speed profiles, a method of curve fitting the sound speed data was chosen. The cubic spline method was used for its simplicity and satisfactory results in smoothly interpolating the sound speed profile data. The cubic spline involves approximating a curve by fitting a third-degree polynomial between each pair of points. The polynomials are computed so that they pass through the given data points and are twice differentiable (Moler, 1970, p. 740). The data points are not required to be equally spaced. Because the resultant curve is an interpolation rather than a smoothing of the data points, the data points are assumed to be free of significant measurement errors.

To compute the spline coefficients, it is assumed that the sound speed data is given as a set of n points such that $c_i = c(z_i)$, $i = 0,1,\ldots,n$. The spline coefficients a_i, $i = 1,2,\ldots,n\text{-}1$, are computed once for a given data set by solving the n-1 simultaneous linear equations

$$\Delta z_i\, a_{i-1} + 2(\Delta z_i + \Delta z_{i+1})\, a_i + \Delta z_{i+1}\, a_{i+1} = \Delta c_{i+1} - \Delta c_i\,,\quad i = 1,2,\ldots,n-1, \tag{2.22}$$

where

$$\Delta z_i = z_i - z_{i-1}, \tag{2.23}$$

$$\Delta c_i = \frac{c_i - c_{i-1}}{\Delta z_i}\quad,\ i = 1,2,\ldots,n, \tag{2.24}$$

and a_0 and a_n are assumed to equal zero. The n-1 linear equations given in Equation (2.22) are tridiagonal[1] in form and can be solved easily using special tridiagonal matrix algorithms (Gerald, 1985, pp. 146-147). Once the spline coefficients are known, the following values are easily computed for a given value of z in the subinterval $z_{i-1} \leq z \leq z_i$ (Moler, 1970, p. 740)

$$c(z) = \overline{w}c_{i-1} + wc_i + (\Delta z_i)^2\, [a_{i-1}(\overline{w}^3 - \overline{w}) + a_i(w^3 - w)], \tag{2.25}$$

$$c'(z) = \Delta c_i + \Delta z_i\, [\, -a_{i-1}(3\overline{w}^2 - 1) + a_i(3w^2 - 1)], \tag{2.26}$$

$$c''(z) = 6\, (\overline{w}a_{i-1} + wa_i\,), \tag{2.27}$$

where

$$w = \frac{z - z_{i-1}}{\Delta z_i} \tag{2.28}$$

and

$$\overline{w} = 1 - w, \tag{2.29}$$

and c(z), c'(z) and c''(z) represent the speed of sound and its first and second derivatives respectively.

[1] Tridiagonal matrices have non-zero elements only on the diagonal and immediately adjacent to the diagonal.

3. Solution of the Ray Equations

With the ability to estimate the sound speed at all depths through the use of cubic splines, the problem of determining the ray path is a matter of numerically solving Equations (2.16) and (2.17) or equivalently, Equations (2.19) and (2.20). Numerical integration, at a most basic level, involves stepping the solution in the independent variable direction (dx) and solving for the subsequent change in the dependent variable (dz and dt) as specified by the numerical integration method being used. Equations (2.16) and (2.17) could easily be solved using an existing numerical integration package (e.g., IMSL). Since the target processor for this ray tracing algorithm was the transputer, it was necessary to write a new numerical integration algorithm. This did however give the opportunity to write an application-specific algorithm that would not require as much overhead as perhaps a general purpose one. Also, it gave an opportunity to investigate different numerical integration algorithms and determine which was most suitable to the ray tracing problem. A comparison of four numerical integration algorithms is given in Chapter III.

III. NUMERICAL INTEGRATION METHODS COMPARISON

A number of numerical integration algorithms were examined and evaluated to see which would be best suited to solving Equation (2.9). They were tested on the basis of their speed, accuracy and ease of implementation. Each method was evaluated using only a simple scenario – a bilinear sound speed profile, a fixed source depth and receiver range. This simple scenario meant that full ray tracing algorithms, able to handle all conditions (i.e., surface and bottom reflections) were not required for each method.

A. NUMERICAL INTEGRATION METHODS

1. General

Four numerical integration methods were evaluated, consisting of two basic types – single step (self-starting) and multistep (predictor-corrector) types. Single step methods are self-starting since they only use information from the previous step. Therefore, to start they only require the initial conditions of the problem and they are able to use different step sizes throughout the integration. Multistep methods take advantage of the information computed in multiple past steps. Therefore, a set of initial conditions is not sufficient to start them. They must be started using single step methods until enough past values are available so they may continue on their own. (Gerald, 1985, p. 312)

The order of a numerical method is related to the amount of accumulated (global) error $E_j[h]$ or the per-step truncation (local) error $e_j[h]$ at a particular step j, where h represents the integration step size. If $E_j[h]$ is $O(h^n)$ then $e_j[h]$ is $O(h^{n+1})$, and the method is considered to be n-th order (Maron, 1982, pp. 340-341). The four methods evaluated are all fourth-order methods.

To solve Equation (2.16) numerically, the solution or ray path, is started at the source depth and range (z_o,x_o) and initial take-off angle (θ_o). The ray path is then stepped in range (x) and an estimate of the new depth (z_{i+1}) is computed by numerical integration. This process is repeated until the receiver range (x_r) has been reached. Note that, in the following discussions of the integration methods, the function f refers to Equation (2.16).

2. Runge-Kutta with Fixed Step Size

The fourth-order Runge-Kutta (RK) method (Gerald, 1985, p. 308) is a single step method which requires four function evaluations, or sample slopes, per iteration. A common Runge-Kutta formula, as applied to the ray tracing problem, is

$$z_{j+1} = z_j + \frac{h}{6}\left\{k_1 + 2(k_2 + k_3) + k_4\right\}, \tag{3.1}$$

where

$$k_1 = f(z_j, x_i), \tag{3.1.1}$$
$$k_2 = f(z_j + \tfrac{1}{2}hk_1, x_i + \tfrac{1}{2}h), \tag{3.1.2}$$
$$k_3 = f(z_j + \tfrac{1}{2}hk_2, x_i + \tfrac{1}{2}h), \text{ and} \tag{3.1.3}$$
$$k_4 = f(z_j + hk_3, x_i + h). \tag{3.1.4}$$

The above equations for $k_1,\ldots,k_4$ represent the sample slopes for various values of z and x, that is $f(z,x)$. The term h refers to the step size or dx, the differential change in range. In the case where the speed of sound varies only with depth (c = c(z)), the function f is only dependent on depth. Therefore the range values are not required and the sample slope equations simplify to

$$k_1 = f(z_j), \tag{3.1.5}$$
$$k_2 = f(z_j + \tfrac{1}{2}hk_1), \tag{3.1.6}$$
$$k_3 = f(z_j + \tfrac{1}{2}hk_2), \text{ and} \tag{3.1.7}$$
$$k_4 = f(z_j + hk_3). \tag{3.1.8}$$

These sample slopes are illustrated in Figure 3.1. The z coordinates of the sample points $P_1, \ldots, P_4$ correspond to z_j, $z_j + \frac{1}{2} hk_1$, $z_j + \frac{1}{2} hk_2$ and $z_j + hk_3$ respectively. The sample slopes $k_1, \ldots, k_4$ are the values of f at the sample points $P_1, \ldots, P_4$.

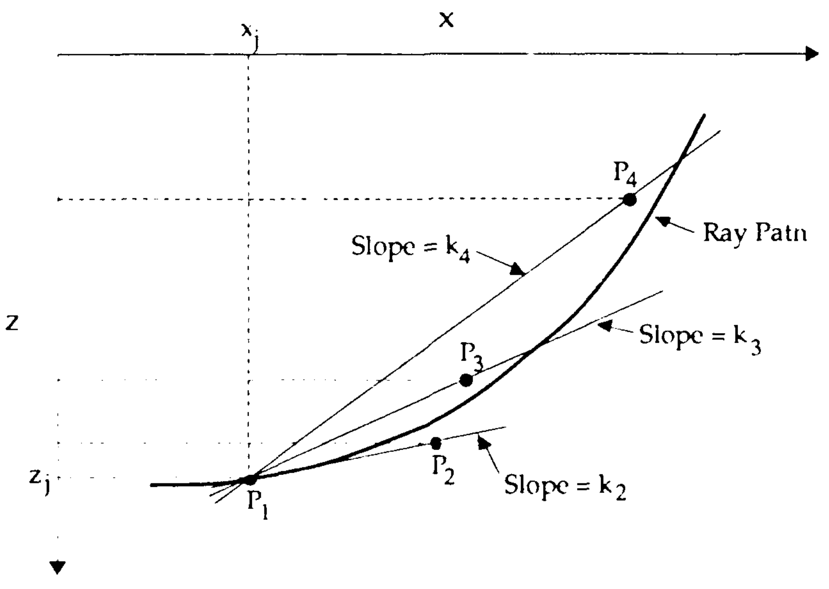

Figure 3.1 **Sample Points and Slopes at (z_j, x_j)**

It is possible to check the accuracy of the RK solution by computing a second estimate of the function with the step size h reduced by one-half. The two solutions can be compared and if a significant difference exists, the integration can be continued using the reduced step size. This approach however requires the calculation of eight more function evaluations per step and represents much more programming effort to implement. An alternative to this approach is to compute more than four sample slopes at each step and use this extra information to yield a second estimate of the solution z_{i+1}. The second estimate of z can be used to adjust the step size up or down depending on whether the quantity dz/dx varies slowly or rapidly (i.e., at turning points) with range. One such method that employs this technique is the Runge-Kutta-Fehlberg algorithm. (Maron, 1982, pp. 348 351) (Gerald, 1985, pp. 309-310)

3. Runge-Kutta-Fehlberg

The Runge-Kutta-Fehlberg (RKF) method is a single-step method with variable step size. The RKF method uses six function evaluations per step and is capable of estimating the error at every step using this extra information. By comparing the error estimate to a fixed error tolerance value, the step size is adjusted up or down accordingly after each step. Although it requires two more sample slopes than the simple Runge-Kutta method, the RKF method is more accurate and can be more efficient because of the step size control. This is illustrated later in Section B – "Test Results."

The RKF equations (Maron, 1982, pp. 350-351), again for the range independent case (c = c(z)), are

$$z_{j+1} = z_j + \frac{25}{216}k_1 + \frac{1408}{2565}k_3 + \frac{2197}{4104}k_4 - \frac{1}{5}k_5, \tag{3.2}$$

$$\text{Error Estimate} = \frac{1}{360}k_1 - \frac{128}{4275}k_3 - \frac{2097}{75240}k_4 + \frac{1}{50}k_5 + \frac{2}{55}k_6, \tag{3.2.1}$$

where

$$k_1 = hf(z_j), \tag{3.2.2}$$
$$k_2 = hf(z_j + \frac{1}{4}k_1), \tag{3.2.3}$$
$$k_3 = hf(z_j + \frac{3}{32}k_1 + \frac{9}{32}k_2), \tag{3.2.4}$$
$$k_4 = hf(z_j + \frac{1932}{2197}k_1 - \frac{7200}{2197}k_2 + \frac{7296}{2197}k_3), \tag{3.2.5}$$
$$k_5 = hf(z_j + \frac{439}{216}k_1 - 8k_2 + \frac{3680}{513}k_3 - \frac{845}{4104}k_4), \text{ and} \tag{3.2.6}$$
$$k_6 = hf(z_j - \frac{8}{27}k_1 + 2k_2 + \frac{3544}{2565}k_3 + \frac{1859}{4104}k_4 - \frac{11}{40}k_5). \tag{3.2.7}$$

The integration step size adjustment is performed in the following manner

- After computing the $k_1,\dots,k_6$ values, the error estimate (Equation 3.2.1) is computed. If the ratio of the error estimate divided by the step size is less than the error tolerance value, then z_{j+1} is computed using Equation (3.2) and the range is incremented by the step size h.

- A step-size scale factor is then computed, regardless of whether z_{j+1} is computed or not, as follows

$$\text{Scale factor} = 0.84 \left(\frac{\text{tolerance} \times h}{\text{error estimate}}\right)^{1/4}. \tag{3.3}$$

- The step size is then scaled (h = Scale factor × h) and the k values computed for the next iteration. Two fixed values representing maximum and minimum scale factors are also used to control how fast the step size can vary at any one time.

The error estimate, Equation (3.2.1), is approximately equal to the per-step truncation error, namely $e_{j+1}[h]$. (Maron, 1982, p. 351)

4. Adams-Bashforth-Moulton with Fixed Step Size

The Adams-Bashforth-Moulton (ABM) method is a multistep or predictor corrector method. It uses four previously computed values to compute a new one and is therefore not capable of self-starting. The ABM equations are (Maron, 1982, pp. 354-356):

the Adams-Bashforth predictor

$$p_{j+1} = z_j + \frac{h}{24}\left\{-9f_{j-3} + 37f_{j-2} - 59f_{j-1} + 55f_j\right\}, \tag{3.4}$$

the Adams-Moulton corrector

$$c_{j+1} = z_j + \frac{h}{24}\left\{f_{j-2} - 5f_{j-1} + 19f_j + 9f_{j+1}(p_{j+1})\right\}, \tag{3.5}$$

and the error estimate

$$\delta_{j+1} = -\frac{19}{270}(c_{j+1} - p_{j+1}). \tag{3.6}$$

The error estimate δ_{j+1} is a measure of the local error $e_{j+1}[h]$ and can be used to determine whether the corrector value c_{j+1} is accurate enough. If not, it can be recomputed using another function estimate, namely the function evaluated using the corrector value (i.e., $f(c_{j+1})$). Although this method is not self-starting, since the $f_{j-3},\ldots,f_j$ are not known, it only requires two or three function evaluations per iteration. A common method of starting (or restarting) is to use the fixed-step-size, fourth-order Runge-Kutta method, slightly modified to save the function values $f_{j-3},\ldots,f_j$ in addition to the $z_{j-3},\ldots,z_j$ computed values. (Maron, 1982, pp. 355-357)

5. Adams-Bashforth-Moulton with Variable Step Size

This is the same method described above with the exception that the step size h can be adjusted in one of two ways based on the error estimate δ_{j+1}. The first is to scale the step size in the same fashion as outlined in the Runge-Kutta-Fehlberg method. However, once the step size is scaled the integration must be restarted. The second approach is to double or halve h only. If the error is unacceptable, then the step can be halved by using the following equations (interpolating quartics) to compute bisecting values

$$f_{j-1/2} = \tfrac{1}{128}\left\{-5f_{j-4} + 28f_{j-3} - 70f_{j-2} + 140f_{j-1} + 35f_j\right\}, \qquad (3.7)$$

$$f_{j-3/2} = \tfrac{1}{64}\left\{3f_{j-4} - 16f_{j-3} + 54f_{j-2} + 24f_{j-1} - f_j\right\}. \qquad (3.8)$$

If the error is acceptable then the step size can be doubled by using every second function value. These step adjustments are depicted in Figure 3.2. For a normal step the new value, f_1, is computed using the four previous values $f_{-3},\ldots,f_1$. When the error is considered too large, the new value, $f_{1/2}$, is computed using $f_{-3/2}$, f_{-1}, $f_{-1/2}$ and f_0 and the step size is reduced by one-half. When the error is acceptable, a new step, f_2, is computed using f_{-6}, f_{-4}, f_{-2} and f_0, thus doubling the step size. Note that doubling the step size requires that seven previous function values must be stored. (Maron, 1982, pp. 357-

359) This method will be referred to as ABMQ, where the "Q" represents the use of the interpolating quartic equations.

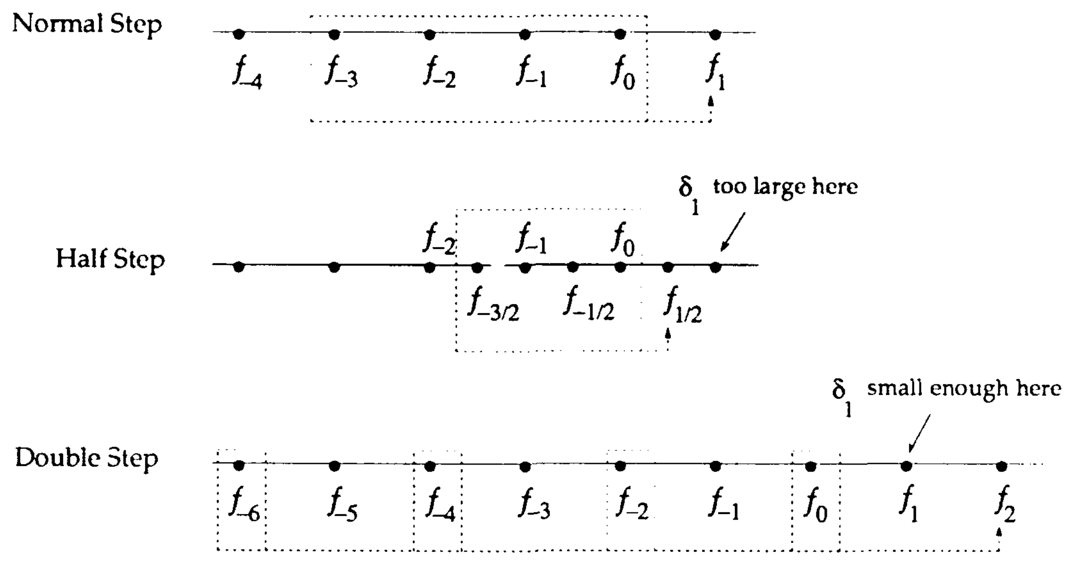
Normal Step
f_{-4} f_{-3} f_{-2} f_{-1} f_0 f_1
δ_1 too large here
Half Step
f_{-2} f_{-1} f_0
$f_{-3/2}$ $f_{-1/2}$ $f_{1/2}$
δ_1 small enough here
Double Step
f_{-6} f_{-5} f_{-4} f_{-3} f_{-2} f_{-1} f_0 f_1 f_2

Figure 3.2 **Adams-Bashforth-Moulton Step Size Adjustment**

B. ALGORITHM TESTING AND RESULTS

1. Test Scenario

The numerical integration methods decribed previously were tested using a simple bilinear sound speed profile as shown in Figure 3.3. The test sound speed profile consists of one gradient of -0.05 s^{-1} from the surface down to 400 meters and another of +0.05 s^{-1} from 400 meters to 800 meters. A source depth of 200 meters was chosen so that a ray leaving the source at 0° would be channeled between 200 and 600 meters and would not interact with the surface or the bottom. This reliable acoustic path (RAP) simplified the support algorithms required to conduct the tests. The ray path is also depicted in Figure 3.3.

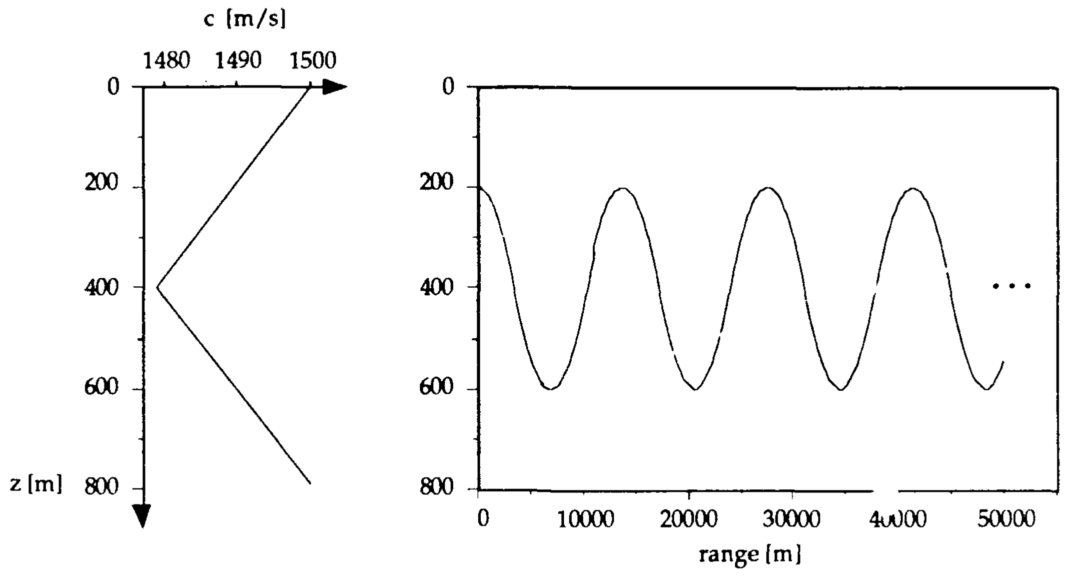

Figure 3.3 **Sound Speed Profile and Ray Path**

All rays were traced out to a range of *100* kilometers which was considered adequate to reveal round-off error accumulation effects. A function to compute the analytic solution of this ray path was also used to compare the results (i.e., the depth) at any given range value. The spline interpolation method was not used in this case so that an analytic solution would be available for comparison.

To determine which numerical integration method was the most efficient in terms of speed and accuracy, a maximum allowable depth error was first chosen. Each algorithm was then run several times to determine which parameters (e.g., step size, error tolerance) were required to satisfy the given allowable error. A maximum error was chosen because the error tended to oscillate with respect to range as illustrated in Figure 3.4. This graph shows the absolute value of the error versus range for the Runge Kutta algorithm with a fixed step size of 200 m. It is clear from this graph that to simply choose the error at a particular range (e.g., receiver range x_r) may give misleading results.

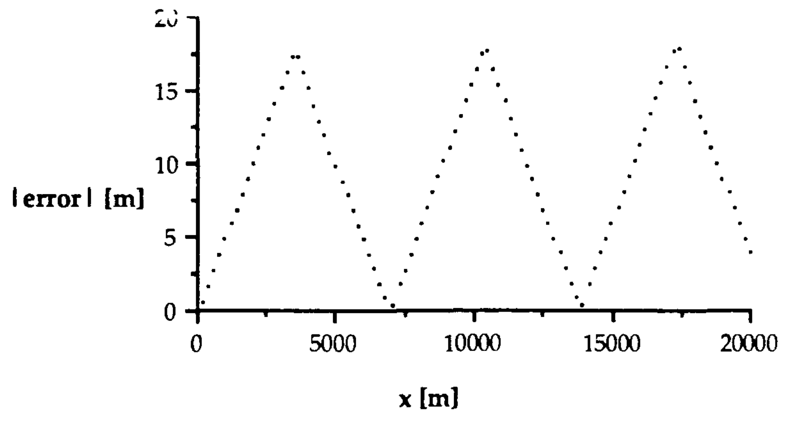

Figure 3.4 **Example Graph of | Error | vs. Range for Runge Kutta with Step Size = 200m**

2. Test Results

The maximum allowable depth error was chosen to be ±5 m. The parameters required by each method are given in Table 3.1 and the normalized timing results are shown in Figure 3.5. It is clear from Figure 3.5 that the RKF method is the most efficient, with an eight-to-one speed advantage over the next closest method (RK).

TABLE 3.1

REQUIRED INTEGRATION PARAMETERS

Method	Step Size [m]	Error Tolerance	Maximum |Error| [m]
RK	45	-	4.86
RKF	-	10^{-5}	4.94
ABM	30	-	4.98
ABMQ	-	10^{-8}	4.98

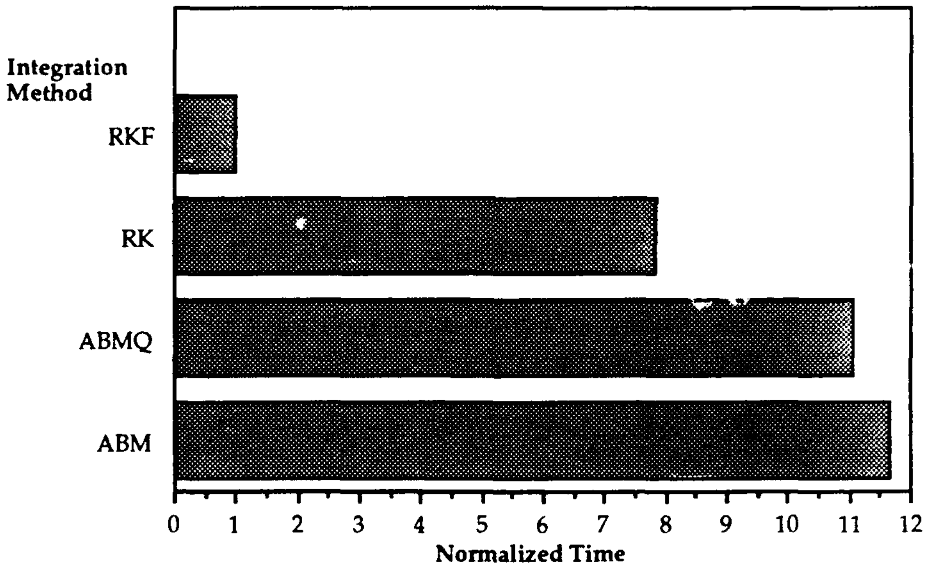

Figure 3.5 **Normalized Timing Results for Maximum Error ±5m**

The advantages of a variable-step method versus a fixed-step method, with regard to the ability to vary the step size and thus control the accuracy, is illustrated in Figure 3.6. This figure shows the range steps taken by the RKF and RK methods out to a range of 5 km. The data used in this graph is taken from the test results described previously; therefore the RK step size is 45m and the RKF error tolerance is 10^{-5}. Evident from Figure 3.6 is the ability of the RKF method to compute the correct ray path but with a far greater efficiency.

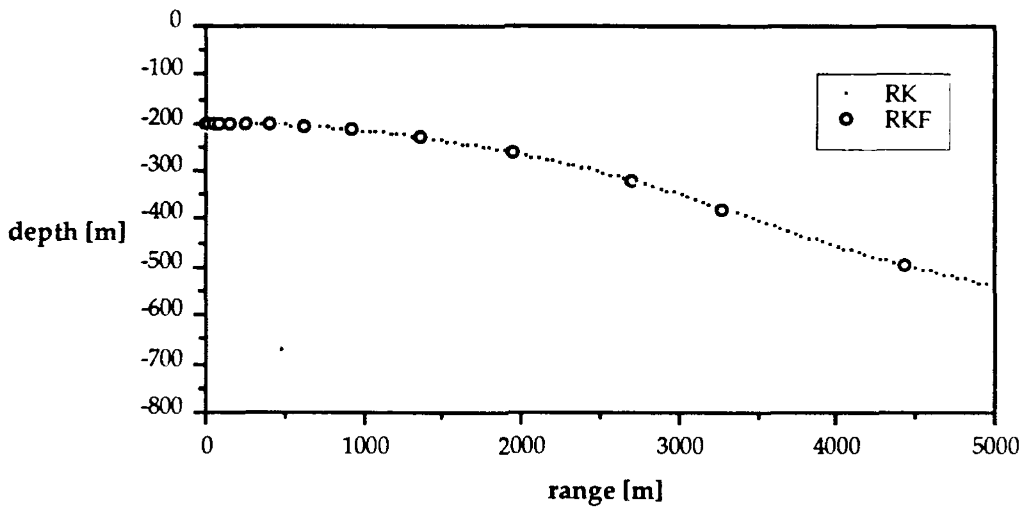

Figure 3.6 **Range Steps Used by RKF and RK Methods**

3. Algorithm Implementation

Before the integration algorithms were tested as previously described, they were implemented and tested using standard examples found in the numerical methods texts (e.g., Maron, 1982, pp. 348-356). Once verified, the algorithms were modified as required so that Equation (2.9) could be solved specifically. Although these modifications were not extensive, it is possible that errors were introduced at this time. This may be the case with the Adams-Bashforth-Moulton methods. The results of the ABM methods indicate that either they are not well suited to the ray tracing problem or else they were not implemented properly. In either case, the ABM methods were much more difficult to implement than the RK and RKF methods. Had the

ABM or ABMQ results been much closer to the RKF results, the RKF method would still have been favored for its much greater ease of implementatior.

C. DISCUSSION

From the results presented, the choice of which numerical integration algorithm to use for the ray tracing problem was quite easy. The Runge-Kutta-Fehlberg method was by far the superior method. By employing two extra equations, compared to the regular RK method, valuable error information can be obtained and used to easily adjust the integration step size. It is the ability to adjust the step size that leads to a much greater efficiency while maintaining the desired accuracy. The RKF algorithm was very similar to the RK with respect to implementation. It could therefore easily replace an existing RK code without significant modifications. Both the RK and RKF methods were very easy to code when compared to the ABM and ABMQ methods.

The ease of implementation is an important issue since the ray tracing codes used in this test only provide minimum capabilities. The full ray tracing code is much more complicated. For example, surface and bottom interactions must be handled and more than one equation must be integrated at the same time. The testing was meant to quickly determine the most suitable numerical integration algorithm for a simple test problem.

IV. GAUSSIAN BEAM TRACING

A. BACKGROUND

1. Intensity Calculations

A common method of estimating the intensity along a particular ray path is to assume that intensity changes are strictly due to spreading loss and that all acoustic energy transmitted between two adjacent rays remains between those rays. The change in intensity is therefore assumed proportional to a change in area. The geometry used in this approximation is illustrated in Figure 4.1. This leads to an equation for the ratio of the intensities, or the focusing factor f

$$f = \frac{I(x)}{I_0} = \frac{x \cos\theta_0}{\sin\theta} \frac{1}{\left|\frac{\partial x}{\partial \theta_0}\right|}, \tag{4.1}$$

where I_0 and $I(x)$ represent the intensities at the source (z_0,x_0) and at a point (z,x) along the ray path respectively. (Brekhovskikh, 1982 p. 40)

A problem occurs with this approach when rays cross, as they do at focusing or convergence zones (Clay, 1977, p. 93). As rays converge, the term $(\partial x/\partial\theta_0)$ approaches zero and the focusing factor approaches infinity. The envelopes of the focusing points, where f equals infinity, are also referred to as caustics. In reality, the intensity increases sharply at caustics but nevertheless, it remains a finite quantity. In order to calculate the focusing factor in this case, modifications to the ray theory must be made.

A formula for the focusing factor on or near a caustic is derived in Brekhovskikh (1980, Sect. 45). The result, as given in Brekhovskikh (1982, pp. 41-42), is

$$f = 2^{5/3} \frac{\cos\theta_o \, (k_o \sin\theta_o)^{1/3}}{\sin\theta} \times \left| \frac{\partial^2 x}{\partial \theta_o^2} \right|^{-2/3} \upsilon^2(t), \tag{4.2}$$

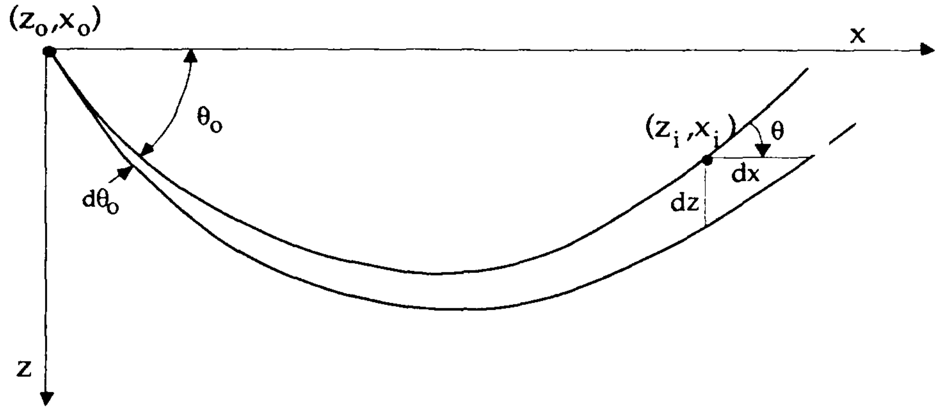

Figure 4.1 **Ray Geometry For Intensity Calculation**

where k_o is the wave number ($k_o = \omega / c_o$) and $\upsilon(t)$ is the Airy function. The argument t of the Airy function is

$$t = \pm\, 2^{1/3} \left| \frac{\partial^2 x}{\partial \theta_o^2} \right|^{-1/3} (k_o \sin\theta_o)^{2/3} (x - x_o), \tag{4.3}$$

where the plus sign is chosen if $(\partial^2 x / \partial \theta_o^2) < 0$ and the minus sign is chosen if $(\partial^2 x / \partial \theta_o^2) > 0$. The condition $t < 0$ corresponds to the interference between rays and thus "spatial oscillations" occur (Brekhovskikh, 1982, p. 42). When $t > 0$, no rays are present and the acoustic field decreases rapidly, as it does at shadow zones. Shadow zones are regions where no rays penetrate; therefore the intensity is assumed zero. Again this is an incorrect assumption since sound does indeed penetrate shadow zones due to scattering, internal waves and diffraction effects (Kinsler, 1982, p. 403).

2. Eigenrays

In order to estimate the multipath arrivals at a given receiver location using ray tracing methods, it is necessary to determine all ray paths which pass through the receiver location. Rays that travel from the source and pass through the receiver location are referred to as eigenrays. Since the ray paths are infinitesimally thin, it is reasonable to set some boundary limits at the receiver location. That is, if a ray passes within a specified distance (e.g., $\pm$ 5 m) from the receiver, it is considered an eigenray. Even with boundary limits set, finding all the eigenrays represents a formidable problem since a multitude of rays must be traced in order to effectively saturate the receiver location with rays. If a full three-dimensional ray trace model (i.e., $c = c(x,y,z)$ and varying bottom bathymetry) is used, the problem becomes very difficult, computationally intensive and can be prone to errors (Porter, 1987, p. 1355).

B. GAUSSIAN BEAMS

1. General

The Gaussian beam (GB) tracing method involves associating "...with each ray a beam with a Gaussian intensity profile normal to the ray." (Porter, 1987, p. 1349) The GB contribution at a particular point can be used to scale other quantities such as intensity or pressure. The problems associated with some ray tracing methods, as described above, are not present in the GB method. The energy at focusing points is finite and shadow zones do contain some sound energy. The GB method also eliminates the need to perform eigenray tracing to compute multipath arrivals.

2. Gaussian Beam Equations

The GB method basically involves solving a set of ordinary differential equations along each ray path. These equations, which are derived from a parabolic equation solution in the vicinity of each ray, are derived in Červený (1982, pp. 109-113) and Porter (1987, pp. 1356-1357). They are related to the width and curvature of the beam associated with a particular ray path. The ray path in this case describes the central axis of the associated beam. These equations are coupled ordinary differential equations and can be solved numerically, along with the ray equation (Equation 2.9), by the methods described in Chapter II. The equations, in terms of the arc length (s) along the ray, are

$$\frac{dq}{ds} = c(s)\, p(s)\,, \tag{4.4}$$

and

$$\frac{dp}{ds} = -\frac{c_{nn}}{c^2(s)}\, q(s), \tag{4.5}$$

where c_{nn} is the second derivative of the sound speed, in a direction normal to the ray. Note that the functions p(s) and q(s) are also complex quantities, that is, $p(s) = p_1(s) + ip_2(s)$ and $q(s) = q_1(s) + iq_2(s)$.

The functions p(s) and q(s) can be related to the beamwidth L(s) and curvature K(s) as follows

$$L(s) = \sqrt{-2/(\omega\, Im\left\{\frac{p(s)}{q(s)}\right\})}\,, \tag{4.6}$$

$$K(s) = -\, c(s)\, \mathcal{Re}\left\{\frac{p(s)}{q(s)}\right\}, \tag{4.7}$$

where $Im\{\}$ and $\mathcal{R}e\{\}$ denote the imaginary and real parts of the complex argument. The beamwidth L(s) is actually the effective beam radius, or the

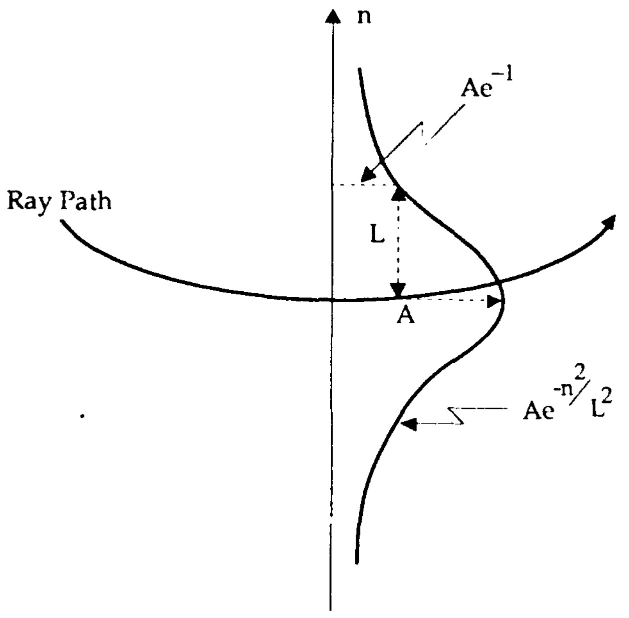

Figure 4.2 **Gaussian Distribution Normal to Ray Path**

width normal to the ray at which the beam amplitude is e^{-1} of its maximum value. (Porter, 1987, p. 1350) This is illustrated in Figure 4.2, which shows the Gaussian distribution in a direction normal to the ray path (Červený, 1982, p. 115).

Once the ray path and the functions p(s) and q(s) are solved, the beam contribution can be computed at any point (s,n)[1] as follows

$$u^{beam}(s,n) = A\sqrt{\frac{c(s)}{x\,q(s)}}\,e^{[-i\omega\,(t(s) + 0.5\,\{p(s)/q(s)\}\,n^2)]}, \qquad (4.8)$$

[1] The coordinate (s,n) represents a point in a ray-centered coordinate system.

where A is an arbitrary constant, t(s) is the travel time along the ray path and n is the distance normal to the ray path. The square root of Equation (4.8) is defined as

$$\sqrt{\frac{c(s)}{x\,q(s)}} = (-1)^{m(s)} \sqrt{\left|\frac{c(s)}{x\,q(s)}\right|}, \tag{4.9}$$

where m(s) is the number of times that q(s) crosses the imaginary axis. (Porter, 1987, p. 1350)

The expansion of a point source into beams is given in Porter (1987, p. 1351). This is necessary because a point source is not "beamlike" and must therefore be approximated by a superposition of beams. The final result of the point source expansion is the following expression for the beam field

$$u(s,n) = \sum_{j=1}^{N\theta} \delta\theta \left\{\frac{1}{c_o}\right\} e^{(i\pi/4)} \sqrt{\frac{q(0)\,\omega \cos\theta_{oj}}{2\pi}}\; u_{\theta_{oj}}{}^{beam}, \tag{4.10}$$

where $\delta\theta$ is the angular spacing between the beams (rays) and $u_{\theta_{oj}}{}^{beam}$ is the beam with an initial beam angle, or take-off angle, of θ_{oj}. The beam field at a point (s,n) is, therefore, the sum of all beam contributions at the same point, multiplied by a beam constant.

The beam equations so far have been presented in ray-centered coordinates. In order to avoid conversions from cylindrical to ray-centered coordinates, an expression for the beam field in cylindrical coordinates is used. The beam field in cylindrical coordinates u(z,x) is equal to u(s,n) as given in Equation (4.10), with the exception of $u_{\theta_{oj}}{}^{beam}$ which is redefined as

$$u_{\theta_{oj}}{}^{beam}(z,x) = A \sqrt{\frac{c(z)}{x\,q(x)}} \exp\left[-i\omega \left\{ t(x) + \frac{1}{c(z)} t_z \Delta z + 0.5\,(\Delta z)^2 \right.\right.$$
$$\left.\left. \left(0.5 \frac{p(x)}{q(x)} t_r{}^2 + 2 \frac{c_n}{c(z)^2} t_z t_r - \frac{c_s}{c(z)^2} t_z{}^2\right)\right\}\right], \tag{4.11}$$

where (t_z,t_r) is the local tangent vector to the ray path and c_n and c_s are the first derivatives of the sound speed in directions normal to and tangent to the ray path, respectively. The local tangent vector (t_z,t_r) is simply equal to $(\cos\theta,\sin\theta)$. The quantity Δz is defined as the distance in the z direction between the receiver depth (z_r) and the ray path at (z,x). (Porter , 1989, p. 2)

3. Initial Conditions

The initial conditions for the ray equation are given by the source position (z_o,x_o) and initial ray angle θ_o. The initial conditions for the functions p(x) and q(x), however, are still an area of current research. Some authors (Červený, 1982) suggest choosing initial conditions so as to minimize the beamwidth at the receiver. This leads to a minimum number of beams needed to describe the field but requires different initial constants for each beam, which may invalidate Equation (4.8) (Porter, 1987, p. 1350). Madariaga (1984) suggests initial conditions that closely follow WKB theory. This approach, however, has p(x) caustics where the beam reduces to a point (Madariaga ,1984, p. 596).

The approach used in this thesis is the one suggested by Porter (1987). They choose initial conditions so that the beams are initially flat (i.e. K(0) = 0) and in the far field the beams are "space filling." Their initial conditions are defined as

$$p(0) = 1 \quad , \quad \text{and } q(0) = i\varepsilon, \tag{4.12}$$

where ε is defined as

$$\varepsilon = \frac{2c_o^2}{\omega\,(\delta\theta)^2}\,. \tag{4.13}$$

4. Beam Contribution Computation

To compute the multipath arrival times and intensities at a given receiver location, a fan of beams (rays) is traced. The contribution of each beam at the receiver location is then computed. These beam contributions can then be used to scale some useful quantity such as the acoustic pressure or intensity. To determine the beam contribution, it is necessary to first determine the ray path segments (ds) whose normals bracket the receiver point at the receiver depth. This concept is illustrated in Figure 4.3.

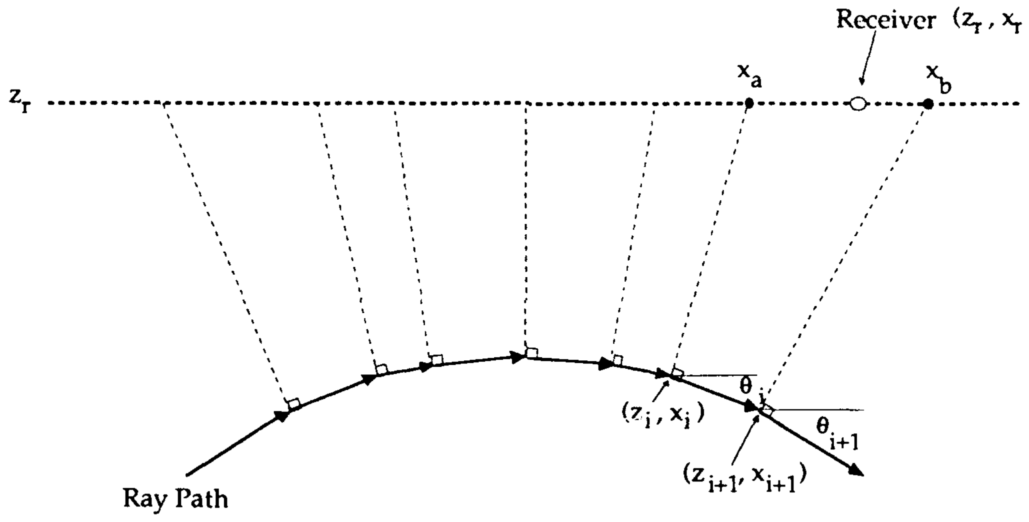

Figure 4.3 **Ray Path Segments and Normal Intercepts**

Figure 4.3 shows that the ray path segments at (z_i, x_i) and (z_{i+1}, x_{i+1}), with angles θ_i and θ_{i+1}, have normals that bracket the receiver location. To compute the intercept (x_{int}) of these normals and the receiver depth line (horizontal dashed line in Figure 4.3), the following formula is used

$$x_{int} = x_i + (z_r - z_i)\frac{\sin\theta_i}{\cos\theta_i}. \qquad (4.14)$$

The ray path segments at (z_i,x_i) and (z_{i+1},x_{i+1}), therefore, have corresponding normal intercepts at x_a and x_b as shown. The following relation yields the proportional distance of the receiver range (x_r) between the points x_a and x_b

$$w = (x_r - x_a)/(x_b - x_a). \qquad (4.15)$$

The contribution at the receiver location can then be approximated by linearly interpolating the necessary quantities z, p(x), q(x) and t(x) along the ray path segment, between the points (z_i,x_i) and (z_{i+1},x_{i+1}), by the same amount. For example, the travel time t(x) used in Equation (4.11) would be computed as $t(x) = t(x_i) + w\{t(x_{i+1}) - t(x_i)\}$. (Porter, 1987, pp. 1352)

To compute the beam field over a large area, a matrix or grid of receiver points is used. The methods described above are valid except that there may be several receiver locations bounded by the ray path normals instead of just one. Also there may be several receiver depths and, therefore, multiple ray normal intercepts. If a continuous wave (CW) source is assumed then the contributions at a particular receiver location are summed over all beams as indicated by Equation (4.10). The results given in Porter (1987) have been computed in this manner. If an impulsive source is assumed, then the contributions are normally kept in discrete form, that is, the contribution information computed at a particular receiver is not summed so that arrival time information is preserved.

5. Reflection at Boundaries

In Porter (1987) only reflections at the sea surface are considered. A beam undergoes a change in curvature (K(s)) but no change in width (L(s)) after a surface reflection. The following conditions are given

$$p' = p + qN, \qquad (4.16)$$

and

$$q' = q\,, \tag{4.17}$$

where the primes denote the quantities after the reflection. The quantity N is defined as

$$N = \frac{4\, c_z\, (\cos\theta)^2}{c^2 \sin\theta}, \tag{4.18}$$

where c_z is the first derivative of the sound speed in the depth direction ($c_z = g$ when $c = c(z)$). Equations (4.16) and (4.17) are considered valid even at grazing incidence. Since the quantity

$$\frac{p'}{q'} = \frac{p}{q} + N \tag{4.19}$$

is related to the beam width and curvature (Equations 4.6 and 4.7), Equation (4.19) indicates that the beam width remains constant and the beam curvature changes during a surface reflection. In this thesis, Equation (4.19) is also used for bottom interactions. (Porter, 1987, pp. 1351-1352)

V. PARALLEL PROCESSING

A. BACKGROUND

Multiprocessor computing systems can offer many advantages over conventional uniprocessor system designs, the most obvious being increased performance. Other benefits include increased reliability, fault tolerance and scalability – the ability to add performance as required. Performance however is not simply a linear function of the number of processors in a system. For example, it is very unlikely that a system with *N* processors will run *N* times faster than a single processor, in solving the same problem. Typically, multiprocessor system performance or throughput, is limited by many factors including interconnection and communication schemes. (Stone, 1987, pp. 278-283)

To solve a problem on a parallel processor architecture first requires dividing the problem up into areas that can be run concurrently. Once this is done, the methods of communication and synchronization between processors must be chosen (Howe, 1987, p. 36). The issue of communication and synchronization, however, is usually determined by the architecture itself and cannot be changed.

1. Granularity

A useful term in describing how a problem or application is broken up into concurrent activities is granularity. Granularity in the context of parallel processing can be defined as "...an indicator of how much computing each processor can do independently in relation to the time it must spend exchanging information with other processors." (Howe, 1987, p. 37) The granularity of an application is referred to as being either coarse or fine-grained. A course-grained application requires much more individual and independent computation time at each processor than the time spent communicating between the processors. A fine-grain application requires less individual and independent computation time at each processor between periods of communications between the processors. (Howe, 1987, p.37)

Another way of looking at granularity is to define two quantities $\mathcal{R}$ and C which represent the time taken in running the computational part of an application and the time taken communicating results between processors, respectively. A coarse-grained application therefore has a relatively high $\mathcal{R}/C$ ratio while a fine-grained one has a relatively low $\mathcal{R}/C$ ratio. (Stone, 1987, pp. 283-284)

2. Communication and Synchronization

Two common methods of processor communications are the shared memory and message passing approaches. In the shared-memory approach, data is communicated between processors by storing it in a common area (memory) where other processors can read it. The message-passing approach is a point-to-point scheme where data generated by a processor is given a destination (address). The data is then passed or routed to the destination. The shared memory approach is analogous to a bulletin board whereas a message-passing approach is analogous to mailing a letter (Howe, 1987, p. 37).

In order to ensure that data is valid, a method of synchronization among communicating processors is required. This is accomplished separately and explicitly in the shared-memory approach through the use of programming constructs such as semaphores and other locking mechanisms. It is up to the programmer to use these constructs, for example, to ensure that data is not read before it is valid. In the message-passing approach synchronization is handled implicitly. That is, if one processor is waiting to perform a communication operation with another processor, they must both be ready before the communication can proceed. This is also referred to as blocked synchronization. An example of this can happen when both processors are at different points in their programs (e.g., one is still computing while the other executes a communications statement). The processor that reaches the communication statement first will be forced to wait until the other processor reaches the same point in its program. (Howe, 1987, pp. 37-38)

B. T800 TRANSPUTER ARCHITECTURE

The T800 transputer is a specialized microprocessor which integrates a 32-bit processor, a floating-point co-processor, 4Kbytes of static RAM, four INMOS communication links with a DMA controller, two timers and a configurable external memory interface on one VLSI device (INMOS, 1987, p. 1). The term transputer is derived from the words **trans**istor and com**puter** and just as transistors are the building blocks of large and sophisticated electronic devices, the transputer was designed to be the building block of distributed computing systems (INMOS, 1986, p. 4). A block diagram of the T800 internal organization is shown in Figure 5.1.

1. Central Processing Unit

The 32-bit central processing unit (CPU) of the T800 consists of instruction processing logic, an instruction pointer, a workspace pointer which points to local variables, an operand register and an evaluation stack consisting of three registers – A, B and C. All instructions refer to the stack implicitly. For example, the ADD instruction adds register A and B and stores the result in register A. As shown in Figure 5.1, the T800 uses three internal buses – a main 32-bit bidirectional address and data bus used by the CPU and the floating point unit (FPU) to access internal and external memory and two unidirectional buses used by the CPU to access the FPU and data links directly (Electronics, 1986, pp. 54-55). The T800 transputer's instruction set follows the load-and-store approach which is characteristic of reduced instruction set computers (RISC) (Gimarc, 1987, pp. 59-63). (INMOS, 1987, pp. 4-5)

2. Floating Point Unit

The T800 contains an integral 64-bit FPU which provides single (32 bit) or double (64 bit) arithmetic and which conforms to the ANSI-IEEE 754-1985 floating point standard. The FPU is microcoded and operates concurrently with, and under control of, the CPU. The FPU evaluation stack

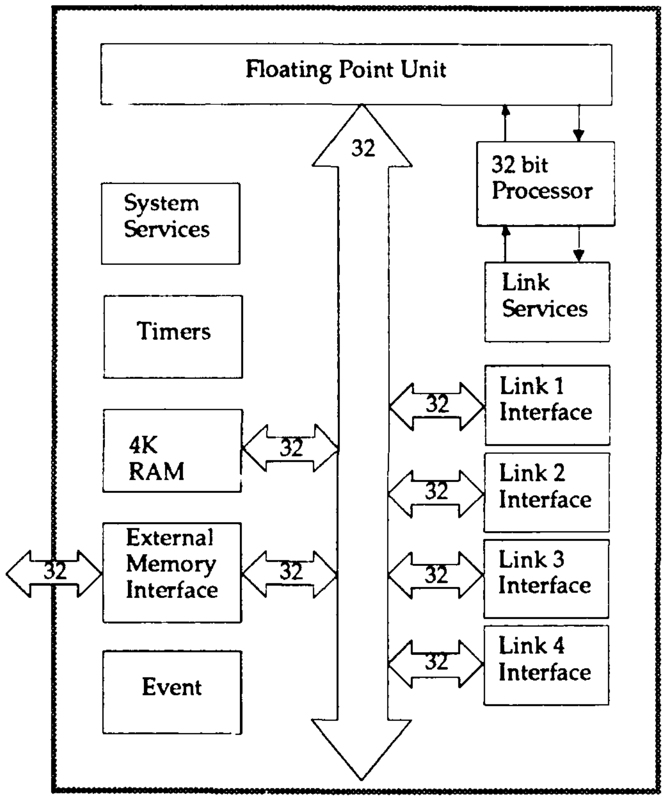

Figure 5.1 **T800 Transputer Block Diagram**

consists of three registers AF, BF and CF which can hold either 32- or 64-bit data. The operation of the FPU evaluation stack is the same as the CPU evaluation stack in terms of load and store effects. Because the CPU and FPU work concurrently, it is possible for the CPU to calculate source and destination addresses for the FPU while the FPU is working on previously supplied data. This is especially important in operations involving arrays of data. Synchronization points are used in the instruction stream wherever data needs to be transferred between the CPU and the FPU. The first processor finished waits for the other to complete its operation; the data is then transferred and both proceed again concurrently. (INMOS, 1987, p. 17)

The T800 FPU incorporates a fast normalizing shifter because of its importance in performing floating point arithmetic and because it was implementable in a reasonable amount of space (silicon). Logic to speed up multiplication and division operations and support square root calculations was also added. Standard mathematical functions (e.g., sin, cos) are implemented using a polynomial approximation method which is slower than some other methods but does not require additional FPU hardware. (INMOS, Tech. Note 6, pp. 7-8)

3. Timers, Processes and Process Scheduling

The T800 contains two 32-bit cyclic timers which allow operations such as reading the time value, delaying execution until a certain time has been reached and timing out for a specified amount of time. One timer is high priority and increments every microsecond and the other low priority timer increments every 64 microseconds.

Processes are one of the fundamental elements of the transputer's model of concurrent processing – the Communicating Sequential Process (CSP) model. The CSP model is a predicate calculus developed by C.A.R. Hoare and has some simple rules

- data may not shared by processes running concurrently,
- all data passing is done through communication, and
- all communication is synchronous.

The CSP model therefore is basically identical to a message-passing architecture with blocked synchronization. (Davidson, 1988, pp. 5-7)

Processes start and run until completion; they can communicate with other processes, spawn other processes and any number of them can be run in parallel. The transputer contains a microcoded scheduler which handles the running of processes in parallel. Note that on a single transputer, processes run in parallel are actually timesliced. The term parallel is still used in this case because the CSP model does not make any assumptions as to where processes are physically running (i.e, on which machines). Processes can be either active (running or waiting on the process queue to run) or inactive (waiting for communication or until a specified time).

Processes are run at either low or high priority with low priority processes running only when there are no high priority processes active. High priority processes are run until completion and are therefore expected to run only for a short time. Multiple high priority processes are run one after another until all are done. If no high priority processes are able to run then the first low priority process on the low priority queue is selected for running. In order to run several in parallel, a low priority process is given two timeslices (approximately 1 msec for each timeslice) to run before being descheduled and put at the end of the low priority queue. Descheduling can only occur during certain instructions, or descheduling points; thereby ensuring that expression evaluation within a process is completed first. Process switching times are typically less than one microsecond. (INMOS, 1987, pp. 6-7)

The CSP model specifies that all communication is synchronous. Logically this means that a process waiting to communicate with another process is blocked until the other is ready. At the hardware level, a process waiting for communication is descheduled until it can proceed with the communication. Because the CSP model makes no assumptions about the underlying hardware, processes may be run in parallel on the same processor, on different processors or a combination of both. In whatever case, the necessary code is the same. Therefore an arbitrarily large system with many parallel processes can be run on a single transputer or many transputers without modification. (Davidson, 1988, pp. 5-7)

4. Communication Links

Communication between processes is achieved through the use of channels and is point-to-point, synchronous and unbuffered. Channels between two processes running on the same transputer are implemented by a word in memory. Channels between two processes executing on different transputers are implemented by physical links. Links consist of two serial, unidirectional wires that can be connected directly between transputers. Link interfaces are TTL-compatible and can therefore be connected directly up to approximately two feet before buffering is required. The link receivers use phase-locked loops to overcome phase differences between the signals of different transputers but are sensitive to skew. The links on the T800 can be configured to run at either 5, 10 or 20 Mbits/sec. All links run at the same speed except Link 0 which can be set independently. (INMOS, 1987, pp. 42-43) (INMOS, 1988, p.19)

Data is communicated as a series of bytes, each of which must be acknowledged before the next is sent. Bytes are sent as 11-bit packets while acknowledgements consist of a start bit followed by a stop bit. The T800 employs overlapped communications so that an acknowledgement can be sent as soon as a data packet has been recognized. This acknowledgement can also be recognized before the the data packet is completely sent, thus allowing the next data packet to be sent immediately after the last one. The communication format is shown in Figure 5.2. Data buffering is provided in the T800 link hardware so that a data rate of 1.74 Mbytes/sec can be achieved in one direction and 2.35 Mbytes/sec when data is sent in both directions at once. (INMOS, Tech. Note 6, pp. 12-13)

Setting up a link transfer requires approximately 20 cycles (1 μsec). The T800 uses an internal eight-channel DMA controller so that, once a link transfer is setup, it can be run autonomously from the processor, only

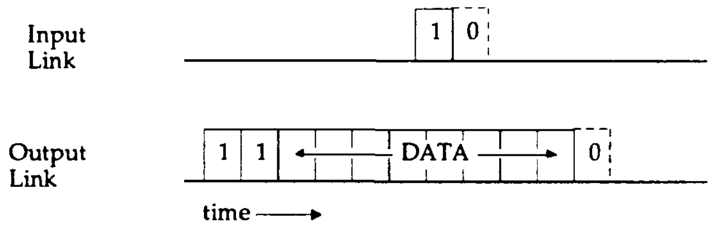

Figure 5.2 **Link Communication Protocol**

requiring one read/write cycle every 32-bit word (usually four processor cycles every 4 μsec). The T800 also uses a double word buffer. The link hardware prefetches the next word to be transferred into the second buffer while outputting from the first. It is possible to run link transfers simultaneously on all four links without seriously degrading the performance of the CPU.

5. Performance

With the T800 transputer running at a clock speed of 20 MHz, the CPU is capable of performing 10 million instructions per second (MIPS) and the FPU capable of 1.5 million floating point operations per second (MFLOPS). Some performance figures for double-precision Whetstones benchmarks are given in Table 5.1 (INMOS, Tech. Note 27, p. 10).

TABLE 5.1

WHETSTONE BENCHMARKS

System	Thousands of Double-Precision Whetstones per second
T800 (20 MHz) (using on-chip, 50 nsec RAM)	4000
MicroVax II (with FPA running MicroVMS)	925
SUN-3 (MC68020 @ 16MHz and MC68881 @ 12.5MHz)	790
VAX 11/780 (8MB memory, FPA, running under UNIX 4.3BSD)	715

C. PARALLEL ALGORITHMS

1. Main Types

Stone (1987) discusses two basic types of physical computational models – the particle model and the continuum model. Problems which are classed as particle models are typically "full-information functions" (Stone, 1987, p. 196). A full-information function is one in which each output quantity depends on all of the input quantities. Examples of this are the FFT and sorting problems. These problems can benefit from parallel processing but are typically difficult to implement. Problems classed as continuum models are much better suited to parallel implementation. When discretized, problems of this form tend to be localized. That is, each point is dependent only on its own state and the states of its adjacent neighbors. Examples of this are convective heat flow and fluid flow. (Stone, 1987, pp. 180-196)

In this thesis we are interested in the ocean acoustic ray tracing problem which can be considered as belonging to the continuum model. In fact the ray tracing problem can be broken up into parts that are completely independent of each other thus simplifying the parallel application further. The parallelization of the ray tracing algorithm is discussed in Chapter VI.

2. Processor Workfarm

a. General

A processor "workfarm" is an approach well suited for implementing some continuum type problems, in particular those that are coarse-grained. A workfarm basically consists of multiple processors or workers, and a controller processor which divides the problem into smaller parts, or work packets, and distributes these work packets. Each worker processor runs the same code and waits for work packets to be sent from the controller. Once the work is completed, the results are returned to the controller and the worker waits for another work packet. The efficiency of a processor workfarm (or any multi-processor system) is dependent on the utilization of the processors. Efficiency is highest when the idle time is minimized; that is, the processors are kept as busy as possible doing useful work.

Note that when data is sent (e.g., work packets, results), it is usually sent in the form of a message. Messages typically contain address information, a message header and the actual data associated with that message. The address, if required, is used to route the message to the correct processor. The message header indicates what type of message is being sent and indicates the type, size, etc of data that follows. Messages and message formats for the ray tracing problem are explained further in Chapter VI.

A typical worker process is shown in Figure 5.3. The outer box represents the *i-th* processor. The circles inside the box represent the processes running in parallel; the arrows represent communication channels and the direction of the communication. The arrows (channels) leaving the box represent physical links while the other arrows inside the box represent internal channels.

b. *Workfarm Processes*

The process called *Throughput* is used to route messages (data) either externally or internally. Messages destined for the next processor are routed through the channel *tonext* and messages for internal (local) use are routed through the channel *tolocal*. The *Render* process is the process that actually performs the computational part of the algorithm. It accepts work packets from channel *frominbuffer* and outputs results through channel *tooutbuffer*. The *Feedback* process multiplexes messages from external processors on channel *fromnext* or internally from the *Render* process on channel *fromlocal*. The messages are then passed up the line on channel *toprev*.

c. *Buffer Processes*

Buffer processes are used between *Throughput* and *Render* and between *Render* and *Feedback*. Because the method of blocked synchronization is used, these buffers ensure that neither the *Throughput* or the *Feedback* processes will be blocked. This situation could arise for example, if the *Render* process was busy performing computations when a new work packet arrived. Without the buffer process, the process *Throughput* would be blocked from sending the work packet to the *Render* process until it finished its work. This could then block subsequent messages destined for other processors, thus degrading the efficiency of the entire system. By using a buffer process, the *Throughput* process is always able to "unload" internal messages and continue routing other messages. The buffer process thus takes over the responsibility of being blocked until the *Render* process is ready for more work. The use of buffering processes can effect the workfarm performance as indicated in INMOS Tech Note 7.

The buffer processes can also be used to buffer (store) work packets. By buffering work packets, the idle time of the render process can be reduced, thus increasing the efficiency. For example, if two buffer processes are used between the *Throughput* and *Render* processes, it would be possible to have three work packets at a given processor, at any one time. The first work packet would be routed immediately to the *Render* process for computation. The second work packet would be routed to the *Render* process but would only get as far as the second buffer process, which would be blocked because *Render* is busy. The third work packet would then only get as far as the first buffer, since the second buffer is blocked. Once the *Render* process finishes its work, it would accept the work packet stored at the second buffer, thus allowing the second buffer to accept the work packet from the first. By having work packets available to the *Render* process immediately, the idle time normally spent waiting for a new work packet to arrive from the controller, is reduced.

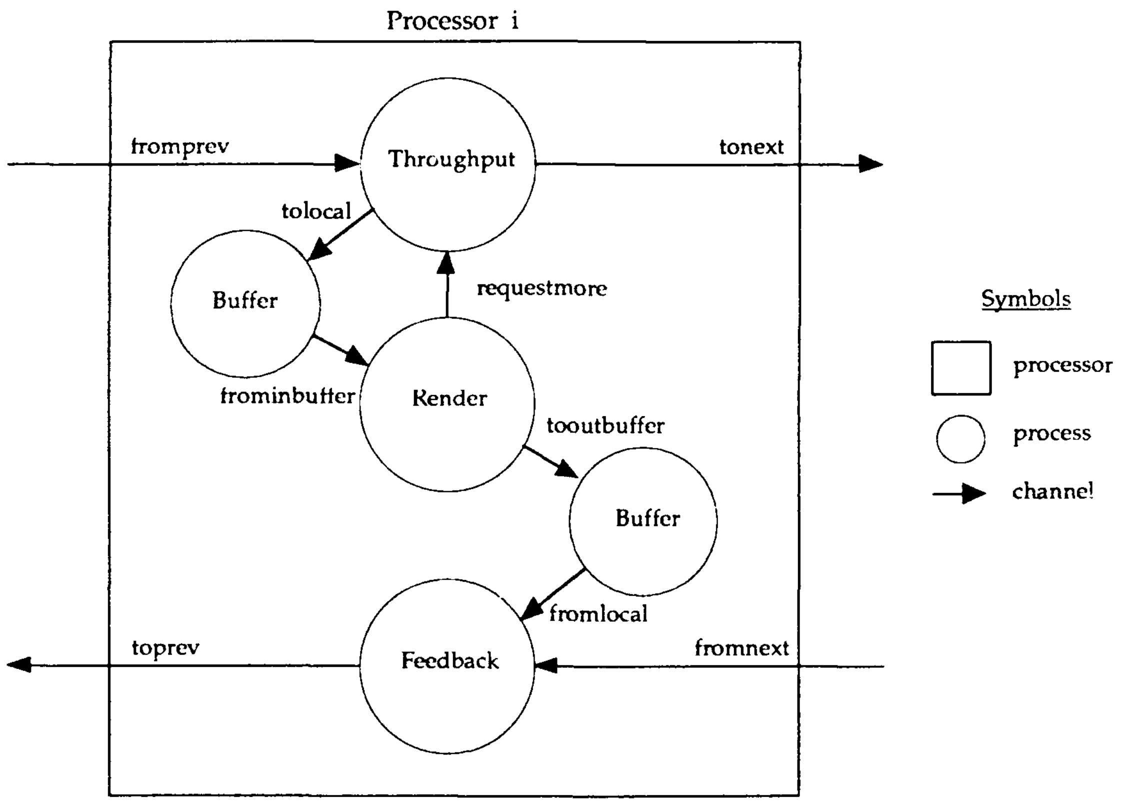

Figure 5.3 **Typical Transputer Workfarm Processes/Channels**

Another method of storing multiple work packets, is to store them at the *Throughput* process. A new channel, *Requestmore* is then used by the *Render* process to indicate that it has finished and request another work packet. The *Throughput* process then sends the work packet to the *Render* process. This method can eliminate the need for multiple buffers between the *Throughput* and *Render* processes but requires storage for the buffered work packets, an additional channel and program logic to handle the requests.

d. Controller Process

The controller function can be performed by a processor in the network (i.e., a transputer) or by a host computer. The controller's task is to break the problem into work packets and distribute them to the worker processors. The controller may also be responsible for receiving and processing the result packets sent by the workers (e.g., graphically displaying results). Work packets can be either addressed or not. If they are addressed, then they are routed to a specific worker processor and, if not, they are handled by the first worker who receives the work packet and is able to handle more work. If n work packets can be buffered by each of the m worker processors then the controller starts by distributing $n \times m$ work packets to the workers. From then on, new work packets are usually not sent out until a result packet has been received. In this way only $n \times m$ work packets are out on the worker network at any given time.

e. Process Priority

Another important aspect of the workfarm is the process priority. As previously stated, processes can be run at either high or low priority with high priority processes running until completion or until blocked. It is a common misconception to think that the computational part of the algorithm should run at high priority and the communication (routing and buffering processes) should be run at low priority. This type of setup however actually leads to decreased performance. The workfarm should be set up so that all communications are run at high priority and computations run at low priority. If the computational part of the algorithm were run at high priority, it would run until completion. Meanwhile, any communication or routing performed at low priority would be completely halted until the computational part was done. This could mean that work destined for other processors would not get through and processors would be idle. Running all communication at high priority ensures that data is never blocked, but rather routed immediately. Since the workfarm approach is best suited to course grain problems, the amount of time required for communication is minimal anyway, compared to the computation time required. Also since communication links, once setup, can run autonomously from the CPU, the computations can be restarted while data is transferred by the link engines. (INMOS, Tech. Note 17, pp. 13-20)

VI. IMPLEMENTATION

A. DEVELOPMENT SYSTEM

1. Hardware

The hardware used for this project included a Macintosh II, a Levco TransLink II transputer motherboard and two TransLink modules. The TransLink II motherboard is a NuBus compatible card that can support up to four TransLink modules. The Macintosh II can hold up to five TransLink II cards for a total of 20 transputers. Each TransLink module consisted of one T800 transputer and 1 MByte of RAM.

2. Software Tools

a. Macintosh

All Macintosh software was written using Symantec's Think C 4.0 development system. Think incorporates a fast compiler, linker, text editor, project organizer and a source level debugger in an integrated environment. Standard ANSI C libraries (e.g., stdio, math, etc.) are included as well as an object-oriented class library for creating Macintosh programs.

b. Transputer

A variety of programming languages and software development systems are currently available for the transputer including C, Pascal, Fortran, or are soon to be released, such as Ada. Of special note is Occam which is a high level language designed specifically to express concurrent algorithms and their implementations on a parallel processing network efficiently and easily. Occam, which was introduced in 1982, is based on the concepts introduced by David May in EPL (Experimental Programming Language) and C.A.R. Hoare in CSP. The development of the transputer was closely linked to Occam and in essence represents an Occam architectural model since many of Occam's constructs are implemented directly in hardware. (*occam® 2 Reference Manual*, Preface)

Software for the transputer was written using Logical Systems Transputer Toolset which includes the software tools necessary to write C programs for the transputer. The tools include a C preprocessor, a C compiler, an assembler, linker and a file librarian. The Transputer Toolset runs under the Macintosh Programmer's Workshop (MPW) shell environment. The C language was chosen over Occam so that all software could be written in one language. This made it possible to write and validate algorithms on the Macintosh before they were ported over and run on the transputer. This was important since it is especially hard to debug parallel software running on a transputer.

B. RAY TRACER

1. Numerical Integration

As discussed in Chapter III, the numerical integration method chosen to solve the ray equation was the Runge-Kutta-Fehlberg method. The algorithm was implemented as presented, but was used to solve not only Equations (2.16) and (2.17), but also the Gaussian beam parameters given by Equations (4.4) and (4.5). All integration was performed with respect to the differential change in range (dx), therefore Equations (4.4) and (4.5) were modified as follows:

$$\frac{dq}{dx} = \frac{c(z)\,p(x)}{\cos\theta}, \tag{6.1}$$

and

$$\frac{dp}{dx} = -\frac{c_{nn}}{c^2(z)\cos\theta}\,q(x). \tag{6.2}$$

Since the functions p(x) and q(x) are complex quantities, they were also separated into their real and imaginary parts which were solved separately. This meant that a total of six equations were numerically integrated simultaneously.

Two variables, *scaleMax* and *scaleMin*, were used to limit the amount by which the step size h could be scaled after each step. These values were typically set to 2.0 and 0.1 respectively. This meant that the step could never be increased by more than twice its previous size or reduced by more than a factor of 0.1, at any one time.

2. Turning Points

When the argument inside the square root in Equation (2.16) approaches zero, this indicates that the ray is approaching a turning point. As the ray turns and changes direction, this argument will ideally equal zero and then begin to increase positively again. However, when solving this equation numerically, it is possible to cause this argument to go negative or equivalently, to step the ray path beyond the turning point. When this happens the algorithm reduces the step size by one-half, the variable *scaleMax* is set to one, thus preventing any further step size increase, and the integration step is started again. This process is repeated until the step size is reduced to a value less than the starting step size *hstart*. In this way, the turning point is approached gradually.

Once the step size is reduced less than *hstart*, an approximation is made to solve for the turning point. The approximation assumes that the sound speed profile is linear at this point and the ray path is defined by the arc of a circle, as discussed in Chapter II. The geometry used for this approximation is shown in Figure 6.1. The point z_i represents the depth value where the approximation is started and the angle of the ray segment at this point is θ. By solving the relation

$$c(z_{i+1}) = c(z_i) + \Delta z g \tag{6.3}$$

for Δz, where $\Delta z = z_{i+1} - z_i$, the value of z_{i+1} (or equivalently z_{tp}) can be computed. Similarly, the relation

$$\Delta x = \frac{1}{ag}(\sin\theta_i - \sin\theta_{i+1}) \tag{6.4}$$

is used to compute the corresponding change in range Δx. (Clay, 1977, pp. 86-87)

The angle at the turning point is zero. The ray path is then stepped past the turning point to z_{i+2}. The depth z_{i+2} is set equal to z_i, the new angle is equal to θ and the range value is incremented by Δx. At this point the numerical integration is restarted using the starting step size *hstart*. The parameter *hstart* was set to 25 m which meant that the corresponding change in depth was very small. Therefore the linear sound speed profile approximation is only used very close to the actual turning point.

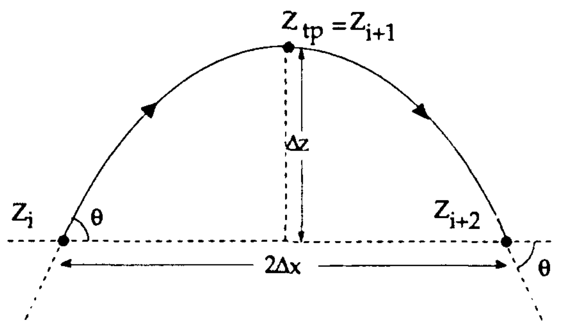

Figure 6.1 **Turning Point Geometry**

3. Surface Reflections

After each integration step the depth value of the new ray path point is tested to see if it is less than zero. If it is, a flag is set to indicate that a surface reflection has been encountered. The step size is then reduced in the same manner described above, and the integration step is repeated. Once again, when the step size is reduced less than *hstart*, the linear sound speed profile approximation is made. The geometry used is shown in Figure 6.2. The ray path is first stepped from z_i to the surface ($z_r = 0$). The angle at the point of reflection (θ_r) is computed using Snell's law and the resultant change in range Δx is computed using Equation (6.4). The ray is then stepped to $z_{i+2} = z_i$ where the new angle is equal to the angle θ at z_i.

4. Bottom Reflections

Bottom reflections are handled in a manner similar to the surface reflections. After each step the depth value of the new ray path point is also

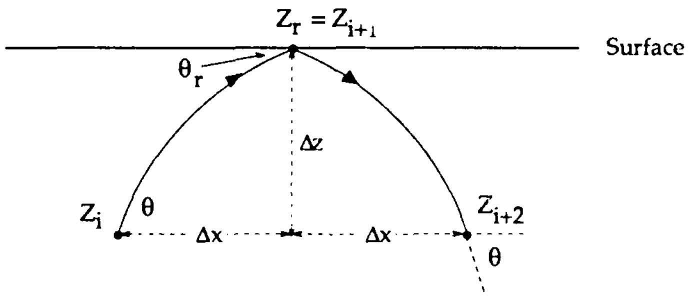

Figure 6.2 **Surface Reflection Geometry**

checked against the bottom depth at that point. If the ray path's depth value is greater than the bottom value, a flag is set indicating that a bottom reflection has occured. The integration step is then repeated with the step size reduced by one-half. This process is repeated until the step size decreases below *hstart* at which point the ray is stepped into the bottom reflection point using the linear sound speed profile approximation method. A bottom tolerance value is used to determine how close the ray is stepped towards the bottom. The bottom depth is provided by a separate function that uses a cubic spline to interpolate bottom bathymetry data. This function takes a range value argument and returns the depth and gradient of the bottom, at that range.

Once the ray path is stepped to the bottom reflection point, it is then reflected out from the bottom. The gradient of the bottom is used to determine the new ray angle and, after a bottom reflection, a new value of Snell's constant (a) must be computed. The geometry used in computing the angle of the reflected ray path is shown in Figure 6.3. The incoming ray has an incident angle *b* with respect to the bottom. The reflected ray will also leave with an angle of *b* when measured with respect to the bottom. However, since all angles are computed with respect to the horizontal this approach must be modified. The incoming ray now has an incident angle *d* and the bottom, at the point of reflection, has an angle *a* with respect to the horizontal. As shown, the reflected ray will have an angle of *c* which is equal to *a+b*. The angle *b* however is equal to *a+d*; therefore, angle *c* is equal to 2*a+d*. The angle *a* of the bottom can be computed from the gradient (g_b) and is equal to $\tan^{-1}(g_b)$.

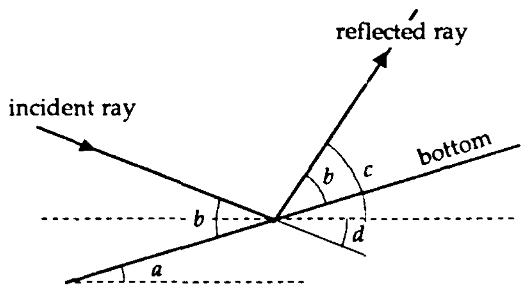

Figure 6.3 **Bottom Reflection Geometry**

It should be noted that there may be cases where the sound speed is required at depths below its maximum tabulated depth. In this case the sound speed is linearly interpolated by the algorithm using a constant gradient of 0.016 s^{-1}. This gradient represents the dependence of the sound speed on depth in isohaline and isothermal water (Clay, 1977, p. 90). If this approximation is not accurate enough, then the sound speed profile should be tabulated to the maximum depth of the bottom. A similar situation arises when the bottom bathymetry is not tabulated out to the desired receiver range. In this case, the bottom, past the last tabulated range value, is assumed flat at a depth equal to the last tabulated depth value. The bottom gradient in this case is equal to zero.

C. GAUSSIAN BEAMS

The implementation of the Gaussian beam contributions was fairly straightforward. A copy of the program used by Porter and Bucker, written in Fortran 77, was provided and used as the basis for the implementation in this thesis. The most notable problems occurred in using complex variables in C. Routines to handle complex math had to be written and complex-valued formulas had to be broken up into their real and imaginary components.

Most of the Gaussian beam algorithm is performed after the ray path has been computed. The following portion of Equation (4.11), however, can be computed at the same time as the ray path

$$\gamma = (0.5\,\frac{p(x)}{q(x)}\,t_r^2 + 2\,\frac{c_n}{c(z)^2}\,t_z t_r - \frac{c_s}{c(z)^2}\,t_z^2). \qquad (6.5)$$

In order to compute the Gaussian beam contribution at a particular receiver location, it is convenient to store the ray path positions, angles, travel times and γ values at all points along the ray. In this way, the ray path normals at each point of the ray are checked to see if they bracket the receiver location. If so, the corresponding angle, travel time and γ values are readily available for use in computing the beam contribution.

D. PARALLEL PROCESSING

1. Parallelization of Ray Tracer

It was stated in Chapter V that the first step in parallelizing a problem is to determine which parts of the problem can be solved concurrently. In the case of acoustic ray tracing, each ray path calculation is independent of all others. Suitable work packets can therefore be formed using individual ray path calculations as the atomic element. That is, work packets can be made up of one or multiple ray path calculations. If a work packet consists of one ray path calculation, then all that is needed to describe it is the initial ray angle. If a work packet is made up of multiple, equally spaced ray path calculations, then an upper and lower angle and an angle increment ($\delta\theta$) are required. Once the individual ray paths have been determined, the Gaussian beam contributions at particular receiver locations can then be summed from the individual ray contributions at the same locations.

In this thesis, work packets are made up of a single ray path calculation and are addressed to a specific processor. A listing of the ray tracing algorithm is given in Appendix A. Note that this listing is for the transputer code and therefore contains the workfarm routines and the ray tracing algorithm.

2. Message Formats

In order to send information to and receive results from the transputers, a number of message formats were devised. Most messages consist of a message header, a processor identification number (address), the length of the message and the associated data. The message header indicates what message is being sent and the processor identification number indicates what processor the message is intended for. The length of the message is the length, in bytes, of the data to follow. The data associated with a message is, of course, dependent on the type of message. Separate messages were used to send sound speed profile data, bottom bathymetry data and other parameters (e.g., integration scale parameters) to the transputers. A listing of the message numbers used is also given in Appendix A.

E. HOST

1. Macintosh Programming

The Macintosh computer, first introduced in 1984, has become known for its ease of use and its unique and consistent graphicai user interface. Most of the software required to support the user interface is contained in ROM (Read Only Memory), which is referred to as the "toolbox". The toolbox contains routines for such things as screen drawing, windows, controls, file manipulation and memory management. The specifications and descriptions of these routines are given in the five volume reference set "Inside Macintosh." By using the toolbox routines where applicable, a consistent user interface can be written for any application. This means that many standard operations are performed in the same manner in all application programs (e.g., saving and printing files) and, as the Macintosh products evolve, applications remain compatible.

Although the end user benefits from the Macintosh software design, programming the Macintosh is a difficult task. For reasons of brevity, the software written as part of the Macintosh host application for this thesis, is not given. However, the basic structure of the main event loop of the host program is given in Appendix B. The software written for the Macintosh includes a version of the ray tracer algorithm to be used in the event that the program is run on a Macintosh without transputers installed.

2. Transputer Interface

All data sent to and received from the transputers is sent as a series of bytes. This is handled by low-level device handling routines on the Macintosh and channel communication routines on the transputer. The ordering of the bytes is reversed between the Macintosh and the transputer; however, the actual byte reversal is handled in hardware on the TransLink card and is transparent to the programmer. The TransLink system is designed so that the Macintosh-to-transputer interface is logically seen as a transputer channel. The Macintosh, therefore, can only communicate directly with one transputer on each TransLink card.

F. OVERALL ALGORITHM SETUP

This section gives a brief description of the ray tracing algorithm. When the program is first started, the host program looks for and boots the transputers. Once the user has set various parameters (e.g., sound speed profile data file, ray angle limits, etc.), and has selected the ray trace command, a number of data structures are sent to the transputers. These include the sound speed profile data, the bottom bathymetry data, the integration parameters and other data required to carry out the ray path calculations. The host program then sends the work packets to the transputers, receives results from them and plots the results on the screen as they are received, until all rays have been traced. If the user chooses to perform another ray trace, perhaps after changing some parameters, the process described above, of sending the intial data structures and work packets, is simply repeated. In the case that no transputers are installed, the program will still function but all computations will be done on the Macintosh II.

VII. RESULTS

A. GENERAL

This chapter presents some results using the ray tracing and Gaussian beam algorithms developed for this thesis and results from the optimization of the parallel processing setup. The first of three example ray traces compares an analytic ray path with that obtained using the algorithm, the second example demonstrates ray interactions with a sloping bottom profile and the third example traces rays for the Munk canonical deep-water sound speed profile. Two Gaussian beam examples, based on the Munk canonical deep-water sound speed profile, are presented as well as plots of the behaviour of the functions used in computing the Gaussian beams. Finally the results of the optimization of the parallel processing scheme are presented.

B. RAY TRACING

1. Comparison with Analytic Example

To demonstrate the accuracy of the ray tracing code, a simple example is presented that can be verified by comparing it to an analytic solution. A linear, upward-refracting sound speed profile was chosen for this example and is shown in Figure 7.1. Because the profile is linear, the ray path follows the arc of a circle whose radius is given by Equation (2.21). To simplify the example further, an initial ray angle of 0° and a source depth of 1000 m was also chosen. The analytic solution yields the following results

$$\text{Snell's constant} = a = 6.83 \times 10^{-4} \text{ s/m}$$

and

$$\text{radius} = \frac{1}{ag} = 9.16 \times 10^4 \text{ m}.$$

The horizontal distance travelled by the ray in any layer is given by (Clay, 1977, p. 87)

$$x_f - x_i = (\text{radius})\,(\sin \theta_i - \sin \theta_f). \tag{7.1}$$

Note that Equation (7.1) has been modified to take into account the fact that

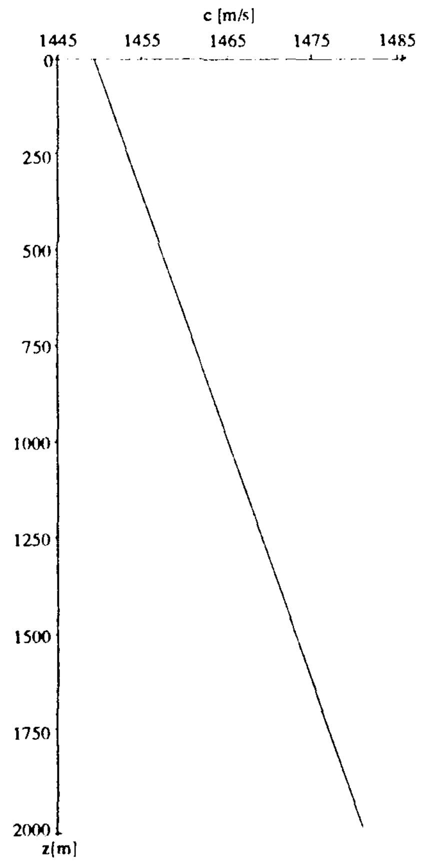

Figure 7.1 **Linear, Upward-Refracting Sound Speed Profile**

all angles used for the ray tracing code are measured with respect to the horizontal and not the vertical as in the reference. Solution of Equation (7.1), with $x_i = 0$ m (i.e. at the source) and x_f the range to the first surface reflection, yields

$$x_f = 1.3495 \times 10^4 \text{ m}.$$

The quantity x_f is also the distance between the sucessive reflection points and turning points, as labelled in the ray path plot of Figure 7.2 (i.e. $x_2 - x_1$ is equal to $x_5 - x_4$, etc.).

A comparison of the results between the analytic and numerical solutions is given in Table 7.1. Note that the numerical solution was computed using an integration tolerance of 1×10^{-6}.

TABLE 7.1

ANALYTIC VERSUS NUMERICAL SOLUTION

Range Point	Analytic Solution [$\times 10^4$ m]	Numerical Solution [$\times 10^4$ m]
x_1	1.3495	1.3495
x_2	2.6991	2.6990
x_3	4.0486	4.0486
x_4	5.3981	5.3981
x_5	6.7477	6.7476
x_6	8.0972	8.0971
x_7	9.4468	9.4466

As can be seen from these results, the ray tracing code is very accurate - differing by a maximum of two meters in the total 100 km range. A smaller integration tolerance would have yielded even higher accuracy.

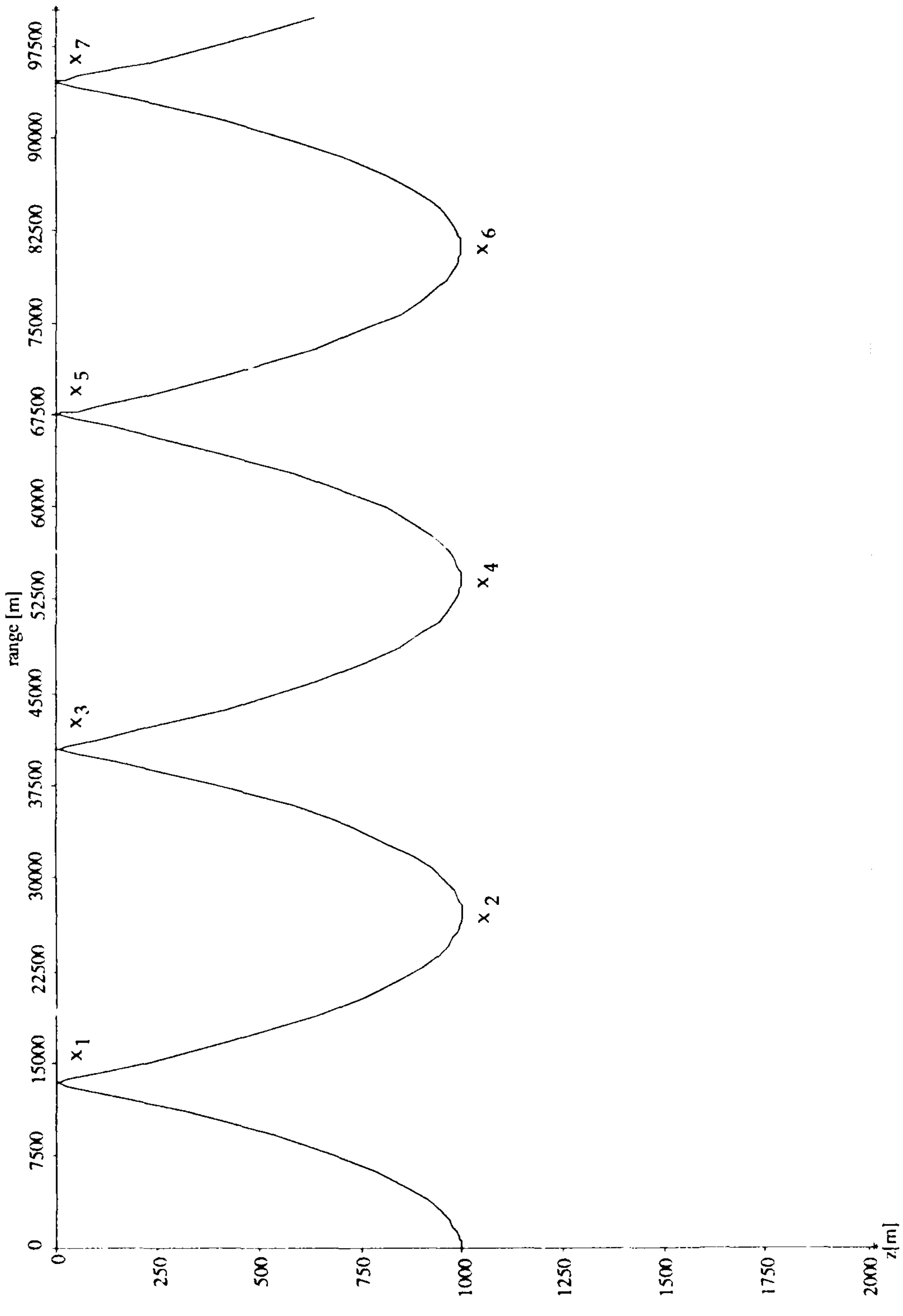

Figure 7.2 **Example Ray Plot**

2. Example with Bottom Bathymetry

The next example demonstrates the ability of the ray tracing code to handle bottom bathymetry data and ray intersections with the bottom. As shown in Figure 7.3, a linear, downward-refracting sound speed profile was chosen so that all rays would be refracted into the bottom.

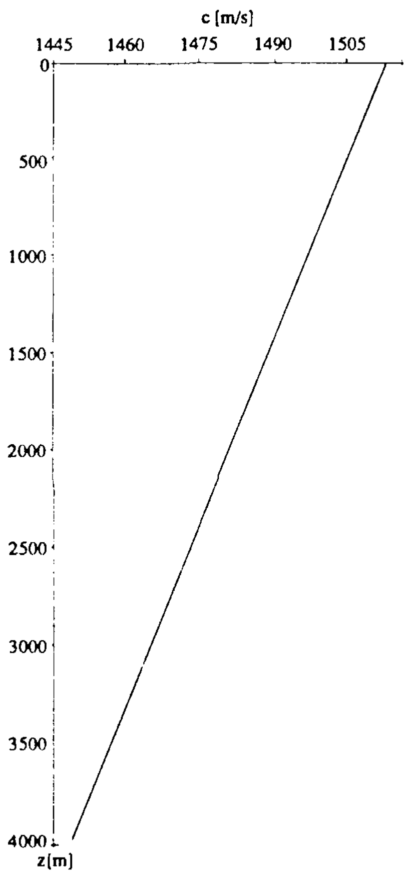

Figure 7.3 **Linear, Downward-Refracting Sound Speed Profile**

The resultant ray plot for a source depth of 3000 m, initial ray angle of -5° and an upward-sloping bottom profile is shown in Figure 7.4. The ray path exhibits behavior as would be expected for rays travelling up a sloped bottom. As the ray propagates up the slope, successive reflections (both bottom and surface reflections) occur closer and closer together. This is due to the fact that when the ray reflects off the bottom, its reflection angle (with respect to the horizontal) is increased by twice the slope of the bottom at the point of reflection (see Figure 6.3). This causes the ray path to 'bunch-up' as it travels up the slope. Under certain conditions, this effect can even cause the ray path to go vertical and then head back towards the source.

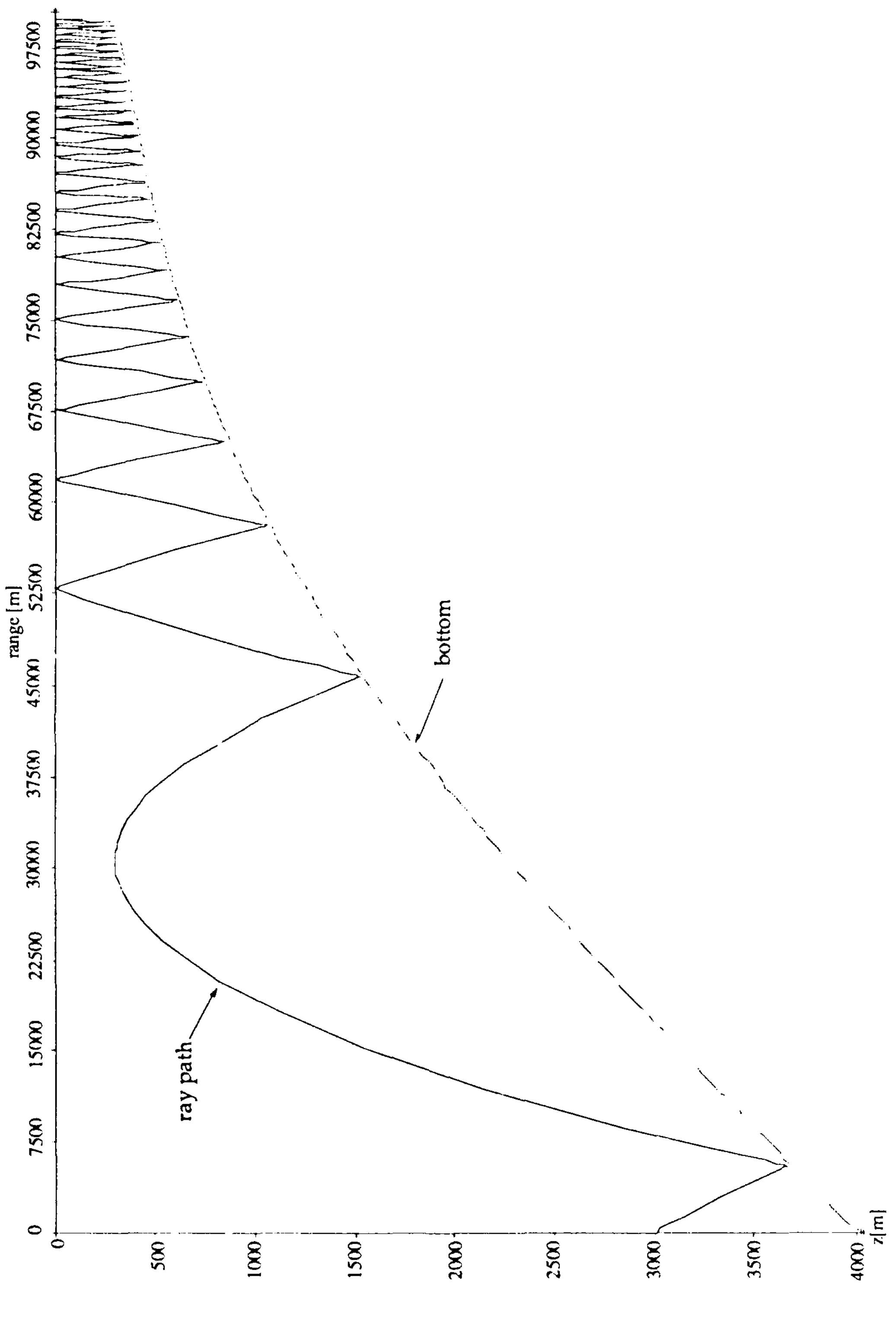

Figure 7.4 **Ray Plot With Bottom Interactions**

3. Munk Profile Example

This example uses a canonical sound speed profile given in Porter (1987), which is referred to as Munk's canonical deep-water sound speed profile. The profile is defined by the equation

$$c(z) = 1500\,\{1.0 + 0.00737[u - 1 + e^{-u}]\}, \quad z \leq 5000\text{m} \qquad (7.2)$$

where

$$u = 2(z - 1300)/1300. \qquad (7.3)$$

A plot of this sound speed profile is given in Figure 7.5 and a ray trace plot using this profile is given in Figure 7.6. Note that in this sound speed profile plot the box symbols represent tabulated or input values. The rays are traced from +14° to -14° with an angular spacing of 0.5° between rays from a source at 1000 m depth. This plot compares favorably to that presented in Porter (1987) and is used in subsequent discussions about Gaussian beam results.

C. GAUSSIAN BEAMS

This section provides some preliminary results obtained using the Gaussian beam algorithm. The results are for single receiver points only (i.e. not total field calculations) and a source frequency of 500 Hz. The ray tracing conditions (i.e. sound speed profile, etc.) are those presented above for the Munk profile example. Note that the term Gaussian beam contribution used in subsequent discussions refers to the fact that at a given receiver location, each ray will contribute to the acoustic field.

1. Example 1 - Receiver at Range 68 km, Depth 1500 m

For the first example, Gaussian beam contributions were computed for a receiver location at a range of 68 km and a depth of 1500 m. This receiver location was positioned away from apparent shadow zones and caustics to provide a more simplified example. The Gaussian beam contributions versus the initial ray angle are shown in Figure 7.7.

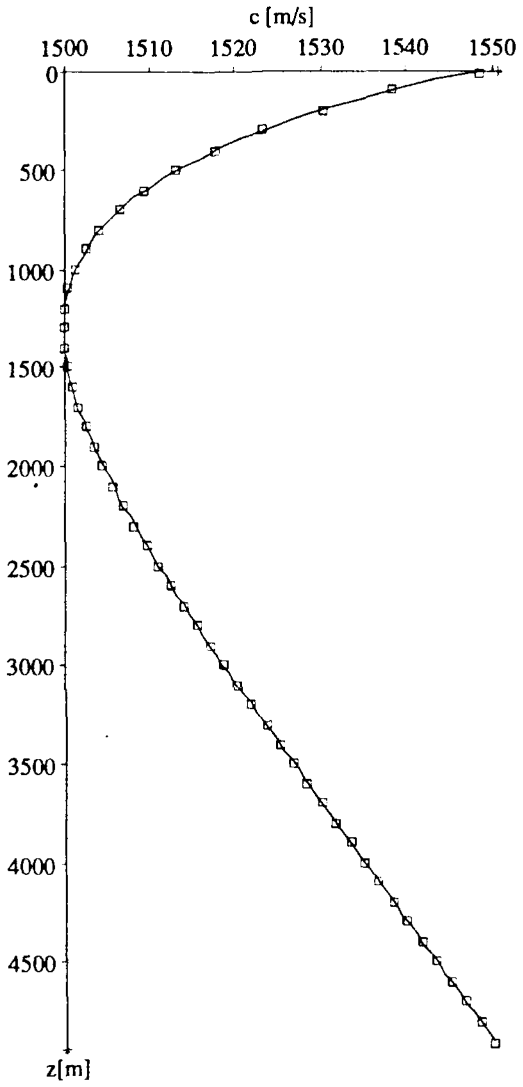

Figure 7.5 **Munk Canonical Deep-Water Sound Speed Profile**

As can be seen from Figure 7.7 a peak occurs at 2.5° (≈ 0.04 radians) and dropouts occur at 13° (≈ 0.23 radians) and at -10° (≈ -0.17 radians). A plot of these three ray paths is given in Figure 7.8.

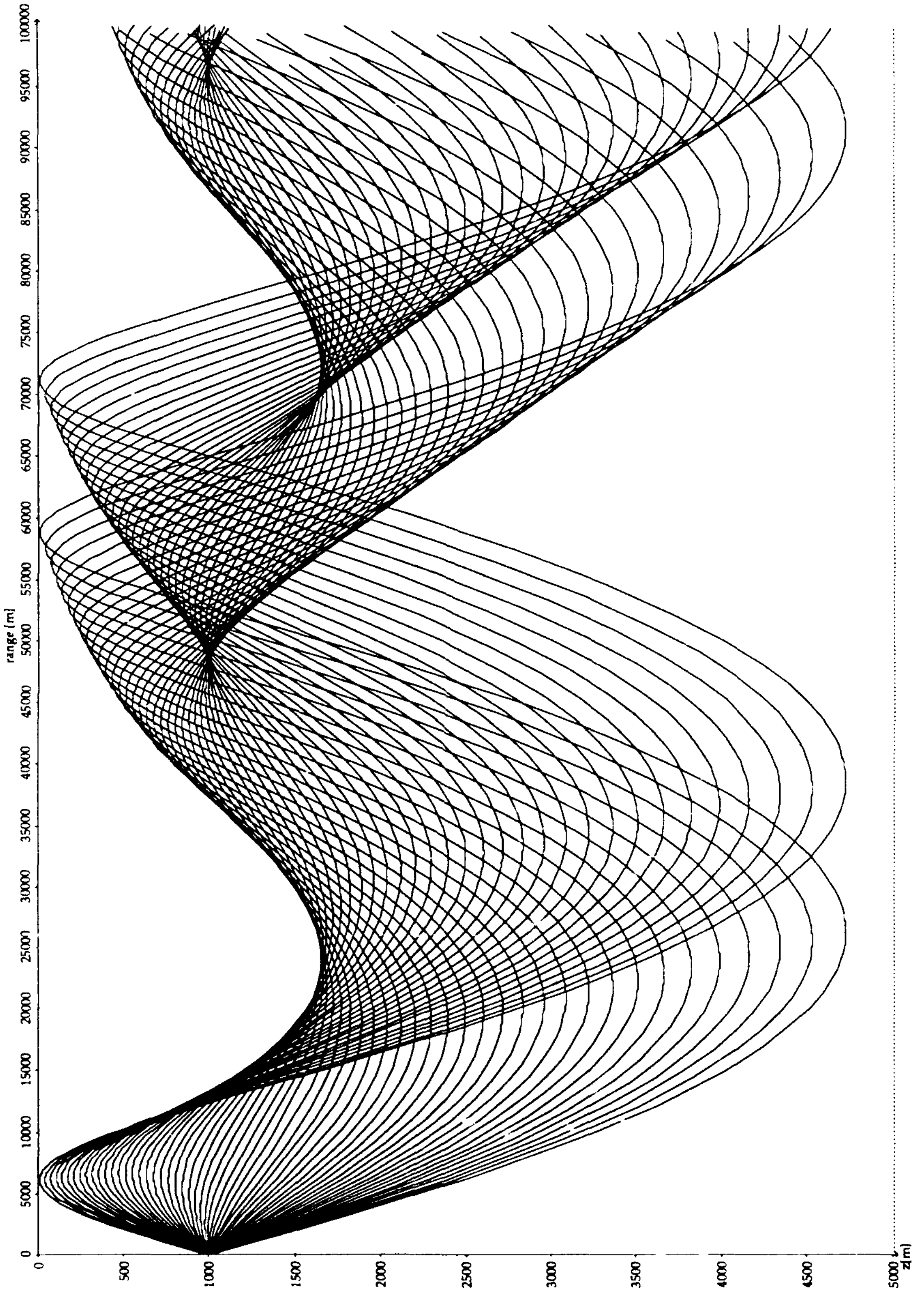

Figure 7.6 **Munk Profile Ray Plot**

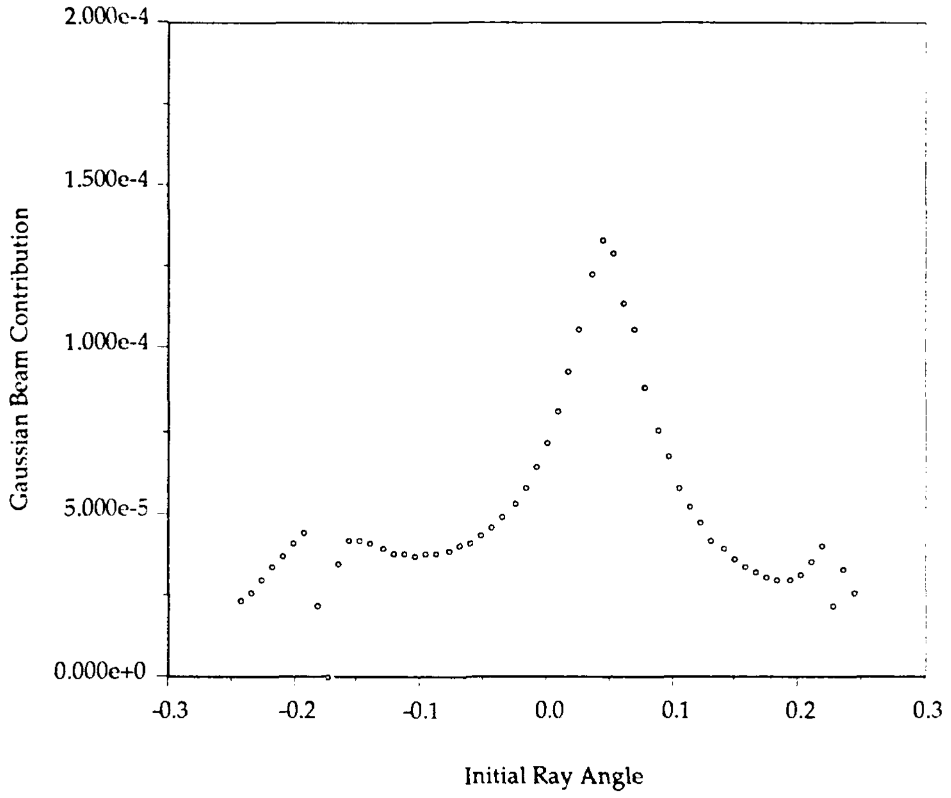

Figure 7.7 **Gaussian Beam Contributions vs. Initial Ray Angle**

A plot of the beamwidth or effective beam radius L(x), as given in Equation (4.6), provides an explanation for the peak and dropout locations. Figure 7.9 below illustrates the behaviour of the Gaussian beam field as a function of the ray path. Figure 7.9 shows a 0° ray path plot with a Gaussian beam field superimposed upon it, reflecting the results of Figure 2 in Porter, 1989. The shaded areas depict the Gaussian beam contribution with the darker areas representing the highest values (i.e. higher intensity, lower transmission loss, etc.). The purpose of this figure is to simply illustrate the focusing and de-focusing effects of the Gaussian beam as a function of the ray path and its effect on contributions at the receiver location. It can be seen in Figure 7.9 that although the ray path passes closer to receiver 2, it would have

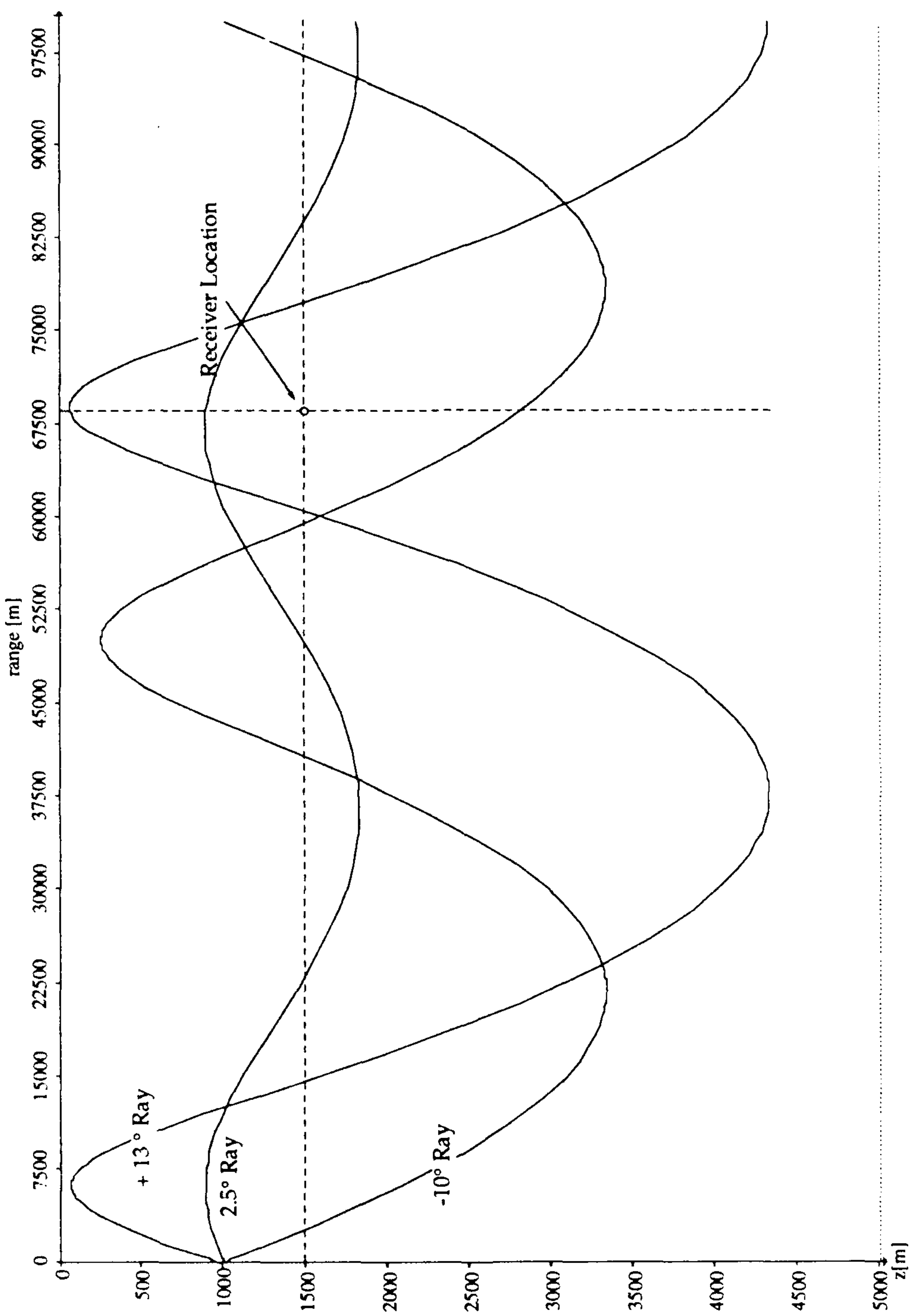

Figure 7.8 **Example Ray Plot**

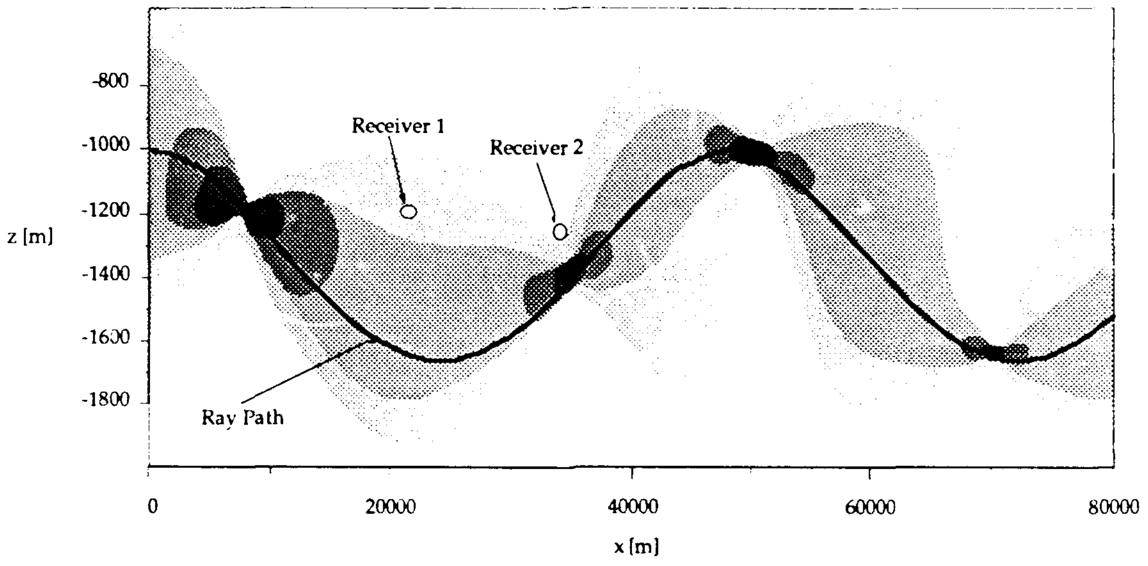

Figure 7.9 **Example Gaussian Beam Field**

a larger Gaussian beam contribution at receiver 1. Therefore receiver 2 is too close to a focusing zone to see any significant contribution from the ray.

It is this focusing and de-focusing effect that is the reason for the dropouts in the Gaussian beam contribution of Figure 7.7. The dropouts at -10° and 13 ° are caused by the fact that the corresponding Gaussian beams are focused and have small beamwidths at the receiver range. The beamwidth of the 2.5° ray is also focused at the receiver location, however not as much as the -10° and 13° rays. This is displayed further in Figures 7.10 through 7.12 which plot the beamwidth L(x) as a function of range (x) for the 13°, 2.5° and -10° ray paths respectively.

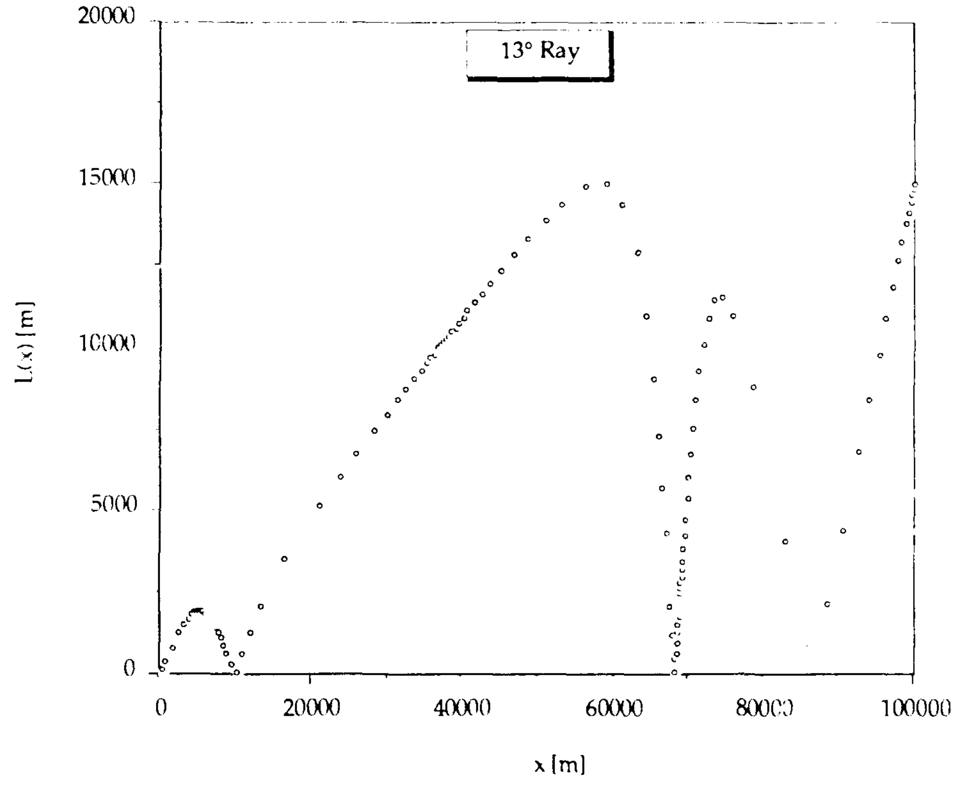

Figure 7.10 L(x) vs. x for 13° Ray

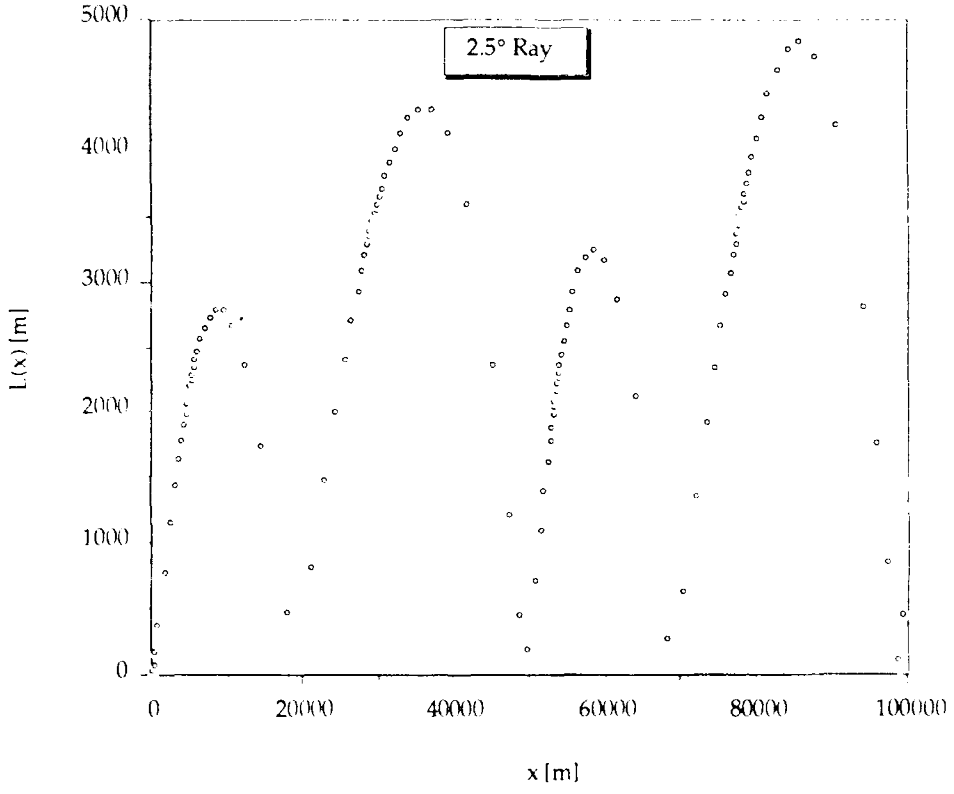

Figure 7.11 **L(x) vs. x for 2.5° Ray**

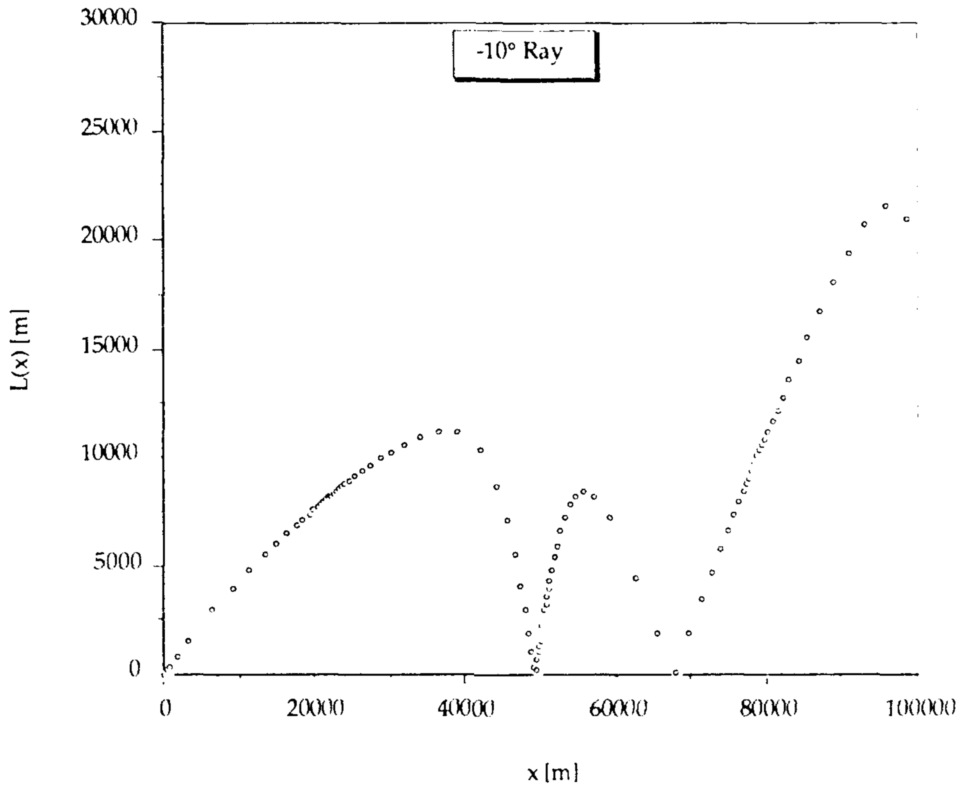

Figure 7.12 **L(x) vs. x for -10° Ray**

2. Example 2 - 5° Ray Path

The second example presented takes a reverse approach in order to confirm the ideas presented above. In this example an arbitrary ray path (other than 13°, 2.5° or -10°) was chosen and a plot of the beamwidth versus the range was made for that ray path. A range at which the beam focused was chosen and the Gaussian beam contributions were then calculated at this range value to see if a dropout existed corresponding to the ray angle.

The ray angle chosen was 5° and a plot of the beamwidth versus the range for this ray is shown in Figure 7.13. The ray focuses at approximately 15.7 km, therefore this range was chosen as a receiver point. Once again the receiver was located at 1500 m depth. A plot of the ray path and receiver location is shown in figure 7.14. A plot of the Gaussian beam contributions at the receiver location is shown in Figure 7.15. It is clear from Figure 7.15 that a dropout occurs at the chosen ray of 5° as expected.

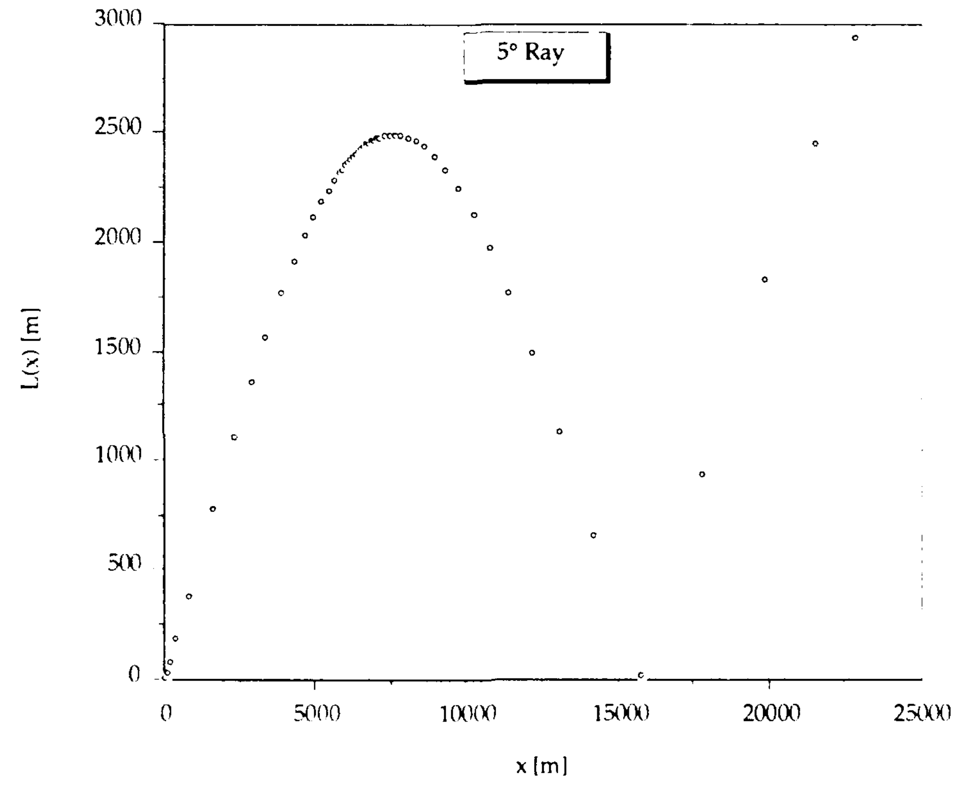

Figure 7.13 **L(x) vs. x for 5° Ray**

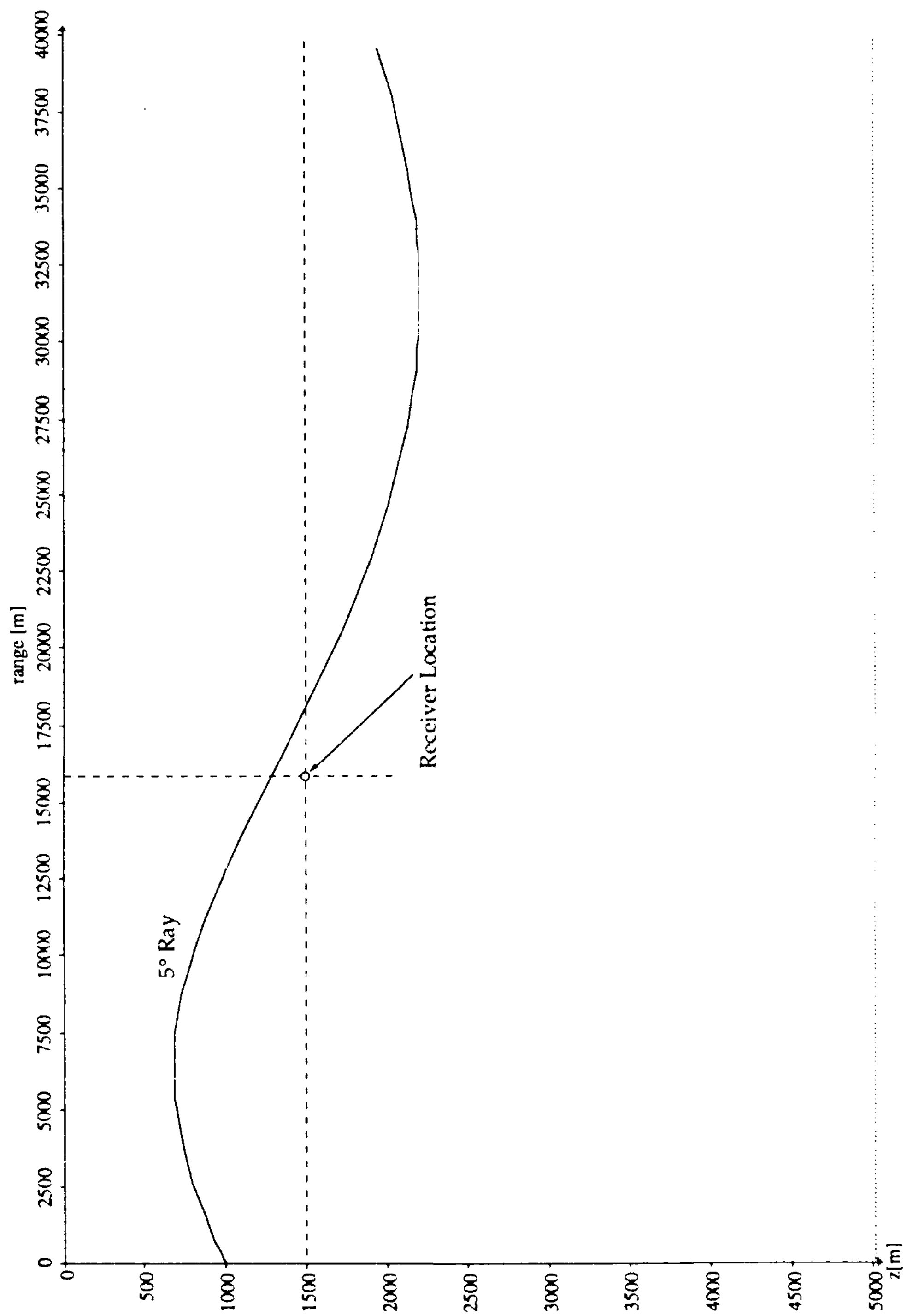

Figure 7.14 **5° Ray Path Plot**

3. Behaviour of p(x) and q(x)

As stated previously in Chapter 4, the quantities p(x) and q(x) (and similarly p(s) and q(s)) are derived from a parabolic equation solution in the vicinity of each ray. They are related to the beamwidth L(x) and curvature K(x) of the beam associated with a particular ray path and are given, in ray

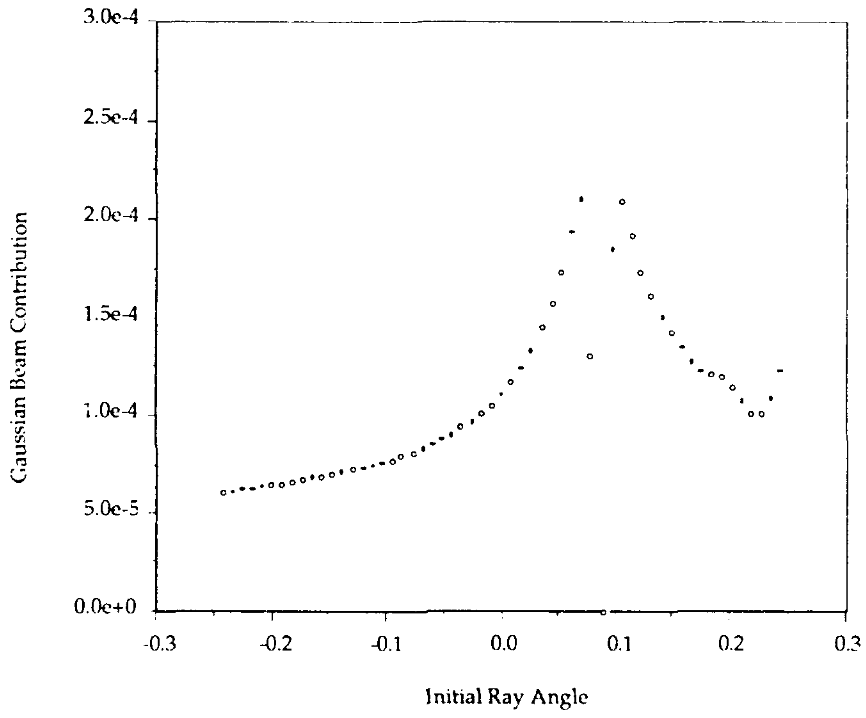

Figure 7.15 **Gaussian Beam Contributions at x = 16.5 km, z = 1500 m**

centered coordinates, in Equations (4.6) and (4.7). The quantities p(x) and q(x) are solved along with the ray equation and are complex functions. Plots of p(x) and q(x) as a function of range are given below in Figures 7.16 and 7.17 respectively for the 2.5° ray. These plots are provided for illustrative purposes since p(x) and q(x) are such fundamental quantities in computing the Gaussian beams. For additional information on the derivation of p(x) and q(x), see Appendix A of Porter, 1987.

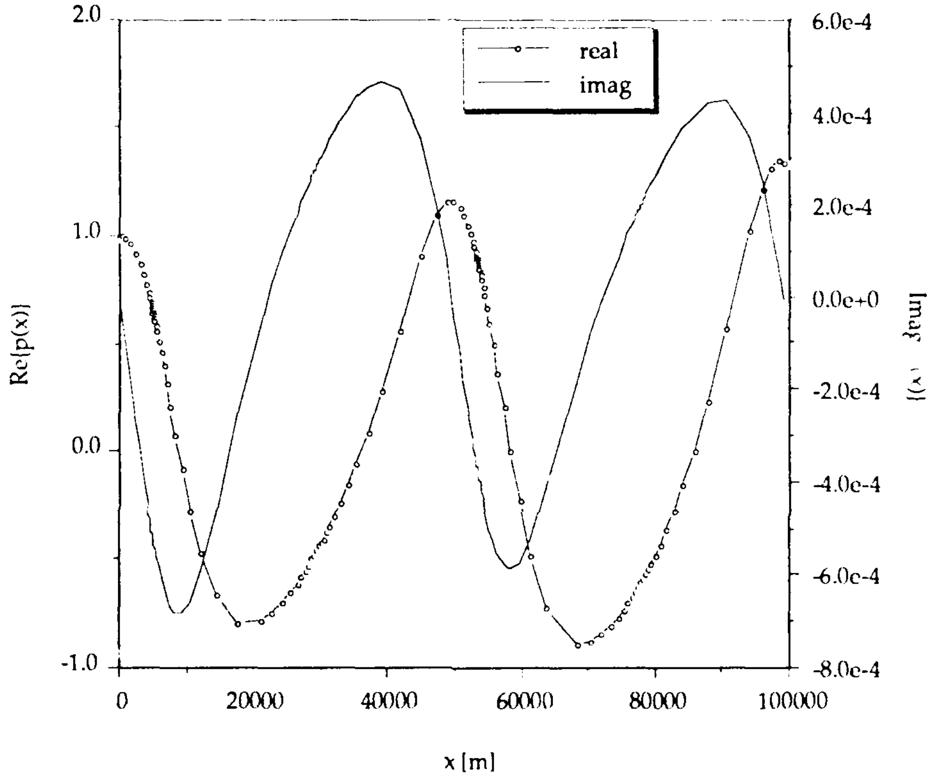

Figure 7.16 **p(x) vs x for 2.5° Ray**

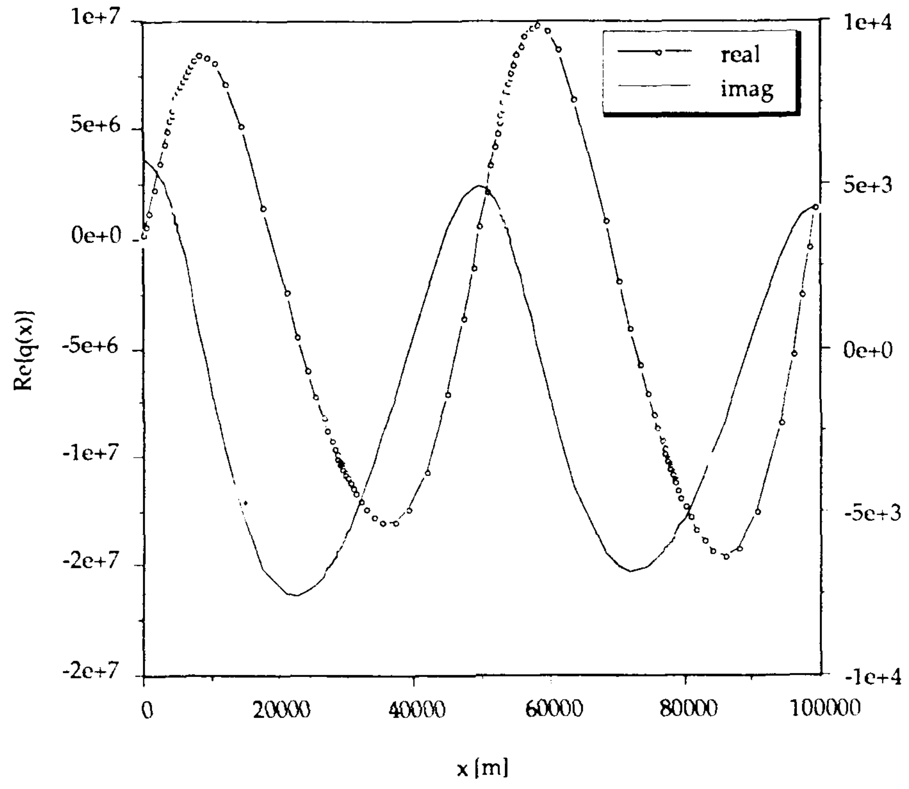

Figure 7.17 **q(x) vs x for 2.5° Ray**

D. PARALLEL PROCESSING

1. General

The parallelization of the ray tracing and Gaussian beam algorithms involved work in many areas including the design and implementation of the workfarm process structure and the implementation of the workfarm and host interface routines. This section presents the results of that effort and results of techniques for optimizing the parallel processing setup. These optimization techniques focused on the use of on-chip RAM and did not involve optimization beyond the intended workfarm concept. Any optimization of the parallel-processor-based Gaussian beam tracing algorithm by using a method other than the parallel processing workfarm is beyond the scope of this thesis and is one of the recommended areas of future work.

2. General Observations

a. Data Reporting

One concern in designing the parallel algorithms was the reporting of data from the transputer network back to the Macintosh II host. Once the transputers had finished a ray path calculation (i.e. the entire ray path) it had to be displayed on the screen and therefore, the ray path data had to be returned to the host. However, this proved to a lengthy process in which any performance gains obtained by using the transputers was degraded in data transfer. Performance was reduced further by the host having to convert ray path data into screen coordinates for plotting. An alternative solution was used whereby the transputer first converted the ray path data into Macintosh screen coordinates (i.e. x and y values). This not only reduced the size of the data that had to be transferred but also reduced the processor workload on the Macintosh II host.

b. ProcAlt Function

Another problem arose in returning results from the transputers to the host. This was caused at the *feedback* process and involved the alternation between the two channels *fromlocal* and *fromnext* (see Figure 5.3). Recall that the channel *fromlocal* accepts internal data and the channel *fromnext* accepts data from the next transputer in the network. The parallel C function *ProcAlt* is used to alternate between input on multiple channels. Its usage and syntax are given in the following example:

```
idx = ProcAlt(fromlocal,fromnext,0);
```

In the example, the function *ProcAlt* checks for any input ready on either of the two channels *fromlocal* or *fromnext*. If input is ready on *fromlocal* it returns a value of zero and if it is ready on *fromnext* it returns a value of one. If neither channel is ready it returns a value of -1. The problem arises when both channels have data ready simultaneously which was the usual case. The function *ProcAlt* tends to favor the channel appearing first in the argument list, in this case *fromlocal*. This meant that any transputers further along in the network would be blocked from returning their results and from doing any more work. In fact, only the first transputer would end up doing any work, thus defeating the purpose of the parallel workfarm.

The solution to this problem was to simply toggle between two separate calls to the *ProcAlt* function on successive passes through the *feedback* process code. The calls were the same except that the channel arguments were reversed in order between the two statements. Therefore the *ProcAlt* function call would still favor the channel appearing first in the argument list but would be forced to alternate between them properly.

c. *Debugging*

Debugging software that runs on a parallel processor workfarm is a difficult process for two main reasons. The first reason is the fact that communications between the host and the network of transputers is performed over a single serial link. This means that any debugging information obtained is usually done using special message formats designed for debugging. It is therefore important to design message formats, etc. to be flexible from the start and with debugging in mind. Some transputer network analyzers do exist but were not available for this thesis work. The second problem also arises from the fact that all inter-transputer communication is performed over serial links and is block-synchronized. As stated previously in Chapter 5, block synchronization means that communication is not performed until both channels are ready. This method of synchronization can lead to a condition known as deadlock whereby one process may be waiting to perform communication with another, but is unable to do so. This process in turn causes another to wait for communication (i.e. block). This condition is repeated throughout until the entire network is in a state of deadlock. Deadlock conditions can either be difficult or easy to solve and is not a problem found in conventional sequential algorithms.

d. Work Packet Buffering

As stated previously in Chapter 5, work packets can be buffered in one of two ways. In this thesis buffer processes were used between the *throughput* and *render* and between the *render* and *feedback* processes. This eliminated both the need to buffer work packets in the *throughput* process and the need for a *requestmore* channel. Different numbers of buffer processes were tried, varying between one and three. In this particular setup, no noticeable differences were observed when the number of buffers was varied. That is, little time was spent waiting for more work to arrive. This is due to the fact that the granularity of the problem was relatively large and the amount of time spent communicating was small in comparison to computational time.

3. Results of Optimization

One of the most effective optimization methods is to maximize the use of the on-chip RAM of the transputer. Although only 4 KBytes of on-chip RAM are available, its 50 nsec access time makes it three times faster than the off-chip RAM (150 nsec). It is therefore best to utilize it whenever possible. Three different memory allocation arrangements were tried. The first was to use off-chip RAM exclusively. The second arrangement used on-chip RAM for the stack space of the *Render* process and the third involved placing certain data structures (variables) and frequently-called routines in the on-chip RAM. The results are given in Figure 7.18. This figure shows the time required to compute each of 11 ray paths, from 5° to -5° with an angular spacing of 1° between rays. The corresponding results for the Macintosh II are also shown.

It can be seen from Figure 7.18 that the use of on-chip RAM for the *render* process stack space gave the best results – at least twice as fast as the Macintosh II. Trying to place some data and frequently called functions in the on-chip RAM seemed to have little effect in speeding up the program. Some performance figures given for the T800 claim that it can perform six times faster than the Motorola 68020/68881 combination, as found in the Macintosh II (Electronics, 1986, p. 52). These figures are usually determined by using program code that is run completely in the T800's on-chip RAM, which has an access time of 50 nsec. In our case, the results for the program run completely in off-chip RAM (150 nsec cycle time) are not quite twice as fast as the Macintosh II. Therefore, assuming that if the program could run completely using on-chip RAM, it would run between four and six times as fast as the Macintosh II version.

The fact that only a factor of two performance gain is realized in any one transputer over the Macintosh II does not seem significant at first. It must be realized however that by using the transputer workfarm, this performance gain can be further multiplied by the number of transputers in the network. Therefore with 20 transputers, a speedup of approximately 40 can be achieved. Of course an upper limit exists on the number of transputers that can be added before other problems, such as communication throughput, actually degrade performance. This upper limit could not be found since additional transputer resources were not available for this thesis.

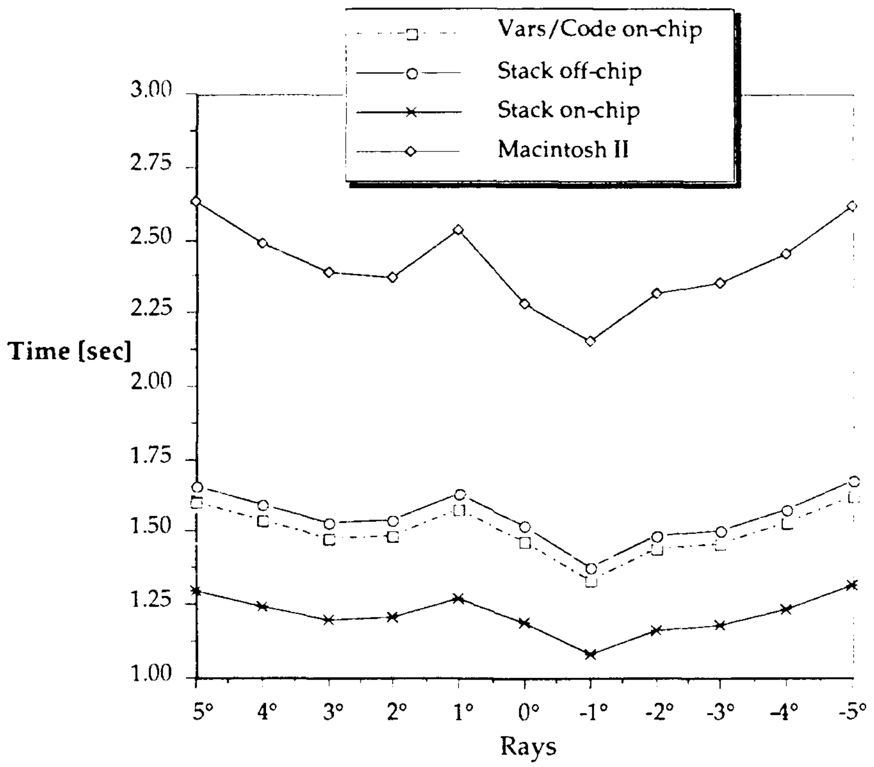

Figure 7.18 **Timing Results**

VIII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

This thesis has attempted to develop a ray tracing model suitable for predicting multipath arrivals and associated information such as travel time and amplitude, for use in ocean acoustic tomography problems.

The ray tracing code developed in this thesis uses the Runge-Kutta-Fehlberg method to integrate the differential equations used in determining the ray paths and associated Gaussian beams. This numerical integration method provides flexibility, accuracy and is more efficient than some other methods. The resultant ray tracing algorithm is also flexible and very accurate.

The method of Gaussian beams is used to estimate the arrival amplitudes at a particular receiver location. This information can be used to scale and estimate other useful quantities such as the pressure or intensity.

This thesis has also shown that it is relatively easy to take advantage of parallel processing without having to buy expensive and specialized equipment and software development tools. The transputer offers a relatively cheap and efficient solution to parallel processing without requiring large amounts of space (20 in one Macintosh II).

B. RECOMMENDATIONS

This section lists some areas and topics related to this thesis that are suitable for further research.

1. Ray Tracing

The model could be made into a full, three-dimensional model. The ray tracing could take into account the range and azimuthal dependency of the sound speed and the bottom could be described using three-dimensional bathymetry. In order to accomplish this, new equations are required for defining the ray path and the Gaussian beams. Also, a suitable method for interpolating the sound speed profile and bottom bathymetry data in three-dimensions is required. In addition to the more complex bottom bathymetry, some form of bottom loss model could also be added to improve the accuracy of the model.

2. Parallel Processing

Only two transputers were used in this thesis to implement the parallel processing workfarm. Additional transputer resources are available at the Naval Postgraduate School and should be used to determine at what point the workfarm speedup peaks (i.e., how many processors). Also, if the model is made more complex (e.g. three-dimensional), it may be necessary to break the ray tracing problem into different parts that can be run concurrently. For example, due to the large amount of information that may be required in a three-dimensional problem (sound speed profiles, bathymetry data), it may be more efficient for one processor to store this information. Other processors would then request (via messages) sound speed and bottom bathymetry information from this processor.

3. Miscellaneous

Although the Gaussian beam contributions were set up to retain travel time information (i.e. an impulsive type source) the ability to compute the total acoustic field, could be added. Also, with the Macintosh II, the addition of color plots would be useful in displaying this field.

APPENDIX A

TRANSPUTER SOFTWARE ROUTINES

This appendix gives a brief description and listing of the software written to run on the transputer. This includes the routines used to implement the workfarm and the routines used to perform the ray path and Gaussian beam calculations.

A. WORKFARM

The routines used to provide the workfarm capabilities are implemented as discussed in Chapter VI. One notable exception is the use of the *buffer* processes. Instead of one *buffer* process on either side of the *render* process, a total of six are used. This still allows work packets to be buffered in a different fashion. Variables to store extra work packets are not required in the *throughput* process and the channel *requestmore* is not required.

B. RAY TRACER

1. Numerical Integration

a. RKF4

This routine represents the core of the ray tracing algorithm. It is used to perform the Runge-Kutta-Fehlberg numerical integration. This routine calls *reducestep* if a turning point or other boundary interaction is encountered.

b. *reducestep*

This routine is called whenever a turning point, surface reflection or bottom reflection is encountered. This routine simply decreases the step size and ensures that the integration step cannot increase further before returning. If the step size is reduced below *hstart*, this routine then calls *turnpoint*.

2. Turning Points

a. *turnpoint*

This routine is called whenever a turning point, surface reflection or bottom reflection is encountered and the step size has been previously reduced less than *hstart*. The routine *turnpoint* handles turning point and surface reflection approximations. Bottom reflections are handled by calling the routine *bottomstep*.

b. *bottomstep*

This routine handles stepping the ray to the bottom and also steps the ray path away from the bottom, using the linear sound speed profile approximation. The ray path is stepped to within a user-specified distance from the bottom.

3. Gaussian Beams

a. *Gamm*

This routine computes a portion of the Gaussian beam formula at the same time as the ray path, as discussed in Chapter VI.

b. *sqrtBranch*

This routine determines whether the function q(s) has crossed the imaginary axis. If it has, then the array index (used to store ray path points) is stored for later use.

c. *GaussSumm*

This routine computes the Gaussian beam summation at a particular receiver point. This routine is called after the entire ray path has been computed.

4. Support Routines

a. *control*

This is the first routine called once a new work packet has been received. It in turn calls the routine *RKF4* to compute the ray path and then the routine *GaussSumm* to compute the Gaussian beam summation. This routine then converts the ray path coordinates (x,y,z) into screen coordinates (h,v) for the Macintosh. The screen coordinate data is then sent to the Macintosh via the buffer processes.

b. *c(z)*

This routine returns the speed of sound, along with its first and second derivatives, at a given depth value. These values are computed using the cubic spline interpolation method.

c. *bottomval*

This routine returns the bottom depth and gradient at a given range value. These values are computed using the cubic spline interpolation method.

d. fcn

This function is used to evaluate the ray equation at the given depth value. The solutions for the travel time and the Gaussian beam equations are also computed.

e. setup

Ths routine converts the initial ray angle from degrees to radians, computes the initial Snell's constant value and sets up the initial conditions for the Gaussian beam solution.

f. tpq

This function computes the travel time and the p(s) and q(s) values using the linear sound speed approximation.

g. increment

This routine simply increments the array index used to store the ray path and Gaussian beam values. It also checks to see if the array index is within the specified bounds.

h. Complex Math Functions

Complex math functions are not handled intrinsically in C, as they are in Fortran. These routines, therefore, provide simple complex math operations such as add, multiply, divide, etc., given two complex arguments.

C. MAIN LISTING

```
/***********************************************************************/

#include <conc.h>       /* concurrency routines for transputer */
#include <math.h>       /* standard math library */
#include "messageID.h"  /* message constants */
#include "raydefs.h"    /* ray tracing constants and structures */

#define TRUE        1
#define FALSE       0
```

```
#define true         1
#define false        0

#define  MAXBUFFER      512
#define  STACKSIZE      1000  /* stack size for Throughput
                              and Feedback*/
#define  CHIPRAMSIZE    3500  /* amount of on-chip RAM used for Render
                              stack frame */

#define getinpacket(c,p,s)    ChanIn(c,(char *)&p,s)
#define putinpacket(c,p,s)    ChanOut(c,(char *)&p,s)
#define getoutpacket(c,p,s)   ChanIn(c,(char *)&p,s)
#define putoutpacket(c,p,s)   ChanOut(c,(char *)&p,s)

int      ourID = 0;           /* our unique ID number */
int      downstream = FALSE;  /* FALSE = no transputers after
                              this one */

/**********************************************************************
*
*     Throughput process of standard workfarm
*
*/
int throughput(procdesc,fromprev, tonext, tolocal)
int procdesc;
Channel *fromprev, *tonext, *tolocal;
{
      int         msg,len;
      int         procID;
      Channel     *chan;
      int         dead,idx;
      char        *buff;
      double      angle;

      dead = FALSE;
      while (!dead) {   /* run until the power is turned off ! */
         do {
            idx = ProcAlt(fromprev,0);     /* wait for something on */
         } while (idx == -1);              /* fromprev channel      */
         switch (idx) {
            case 0:     /* fromprev */
               msg = ChanInInt(fromprev); /* get message type */
               switch(msg){
                  case MSG_PASS:    /* used for passing boot code
                                    downstream */
                     procID = ChanInInt(fromprev); /* get processor ID */
                     len = ChanInInt(fromprev); /* length of message */

                     /* if downsteam = FALSE then pass only data
                     of message ie. the boot code */
                     if (downstream == FALSE){
                        passdata(fromprev,tonext,len);
                        downstream = TRUE;
                     }
                     else{    /* pass the whole thing */
```

```
          ChanOutInt(tonext,MSG_PASS);
          ChanOutInt(tonext,procID);
          ChanOutInt(tonext,len);
          passdata(fromprev,tonext,len);
       }
       break;
    case MSG_PROCID:
       procID = ChanInInt(fromprev);
       if(ourID == 0)
          ourID = procID;    /* we now have an ID number */
       else{     /* we already have an ID */
          ChanOutInt(tonext,MSG_PROCID);
          ChanOutInt(tonext,procID);
       }
       break;
    case MSG_ANGLE:    /* work packet */
       procID = ChanInInt(fromprev);
       len = ChanInInt(fromprev);
       ChanIn(fromprev,&angle,len);
       if(procID == ourID)   /* message is for us */
          chan = tolocal;
       else                  /* message is not for us */
          chan = tonext;
       ChanOutInt(chan,MSG_ANGLE);
       ChanOutInt(chan,procID);
       ChanOutInt(chan,len);
       ChanOut(chan,&angle,len);
       break;
    case MSG_WORK:     /* work parameters */
       procID = ChanInInt(fromprev);
       len = ChanInInt(fromprev);
       buff = (char *)malloc(len);
       ChanIn(fromprev,buff,len);
       if(procID == ourID)
          chan = tolocal;
       else
          chan = tonext;
       ChanOutInt(chan,MSG_WORK);
       ChanOutInt(chan,procID);
       ChanOutInt(chan,len);
       ChanOut(chan,buff,len);
       free(buff);
       break;
    case MSG_TEST:     /* test message */
       procID = ChanInInt(fromprev);
       len = ChanInInt(fromprev);
       if(procID == ourID)
          chan = tolocal;
       else
          chan = tonext;
       ChanOutInt(chan,MSG_TEST);
       ChanOutInt(chan,procID);
       ChanOutInt(chan,len);
       break;
```

```
                  default:     /* used to receive all other message
                               types - they are stored as a series
                               of bytes */
                     procID = ChanInInt(fromprev);
                     len = ChanInInt(fromprev);
                     buff = (char *)malloc(len); /* allocate space */
                     ChanIn(fromprev,buff,len);  /* read data */
                     if(procID == ourID)
                        chan = tolocal;
                     else
                        chan = tonext;
                     ChanOutInt(chan,msg);
                     ChanOutInt(chan,procID);
                     ChanOutInt(chan,len);
                     ChanOut(chan,buff,len);
                     free(buff); /* free memory */
                     break;
                  }
               break;
            default:
               while (!dead) {
                  /*msg = ChanInChar(fromprev);*/ /* hang */
               }
               break;
         }
      }
}

/************************************************************************
*
*     This routine is used by the throughput process to pass
*     data through from an input channel to an output channel.
*     it does so in blocks of up to MAXBUFFER bytes.  Taken from
*     examples provided by Levco.
*/
void passdata(inc, outc, len)
Channel *inc, *outc;
int len;
{
      char buffer[MAXBUFFER];
      int todo, thislength;

      todo = len;
      while (todo > 0) {
         thislength = todo>MAXBUFFER ? MAXBUFFER:todo;
         ChanIn(inc,buffer,thislength);
         ChanOut(outc,buffer,thislength);
         todo -= MAXBUFFER;
      }
}
/************************************************************************
*
*     This is the standard buffer process. Note that although
*     six buffers are used, only one copy of the code is
*     required.
```

```
*/
int buffer(procdesc, toBuffer, fromBuffer)
int procdesc;
Channel *toBuffer, *fromBuffer;

{
      int msgID, procID, msgLength;
      char *msg;

      while (TRUE) {
         msgID = ChanInInt(toBuffer);
         procID = ChanInInt(toBuffer);
         msgLength = ChanInInt(toBuffer);
         if (msgLength != 0){        /* if message length = 0 then
                                        ChanIn will not work !! */
            msg   = (char *)malloc(msgLength);
            ChanIn(toBuffer, msg, msgLength);
         }

         ChanOutInt(fromBuffer, msgID);
         ChanOutInt(fromBuffer, procID);
         ChanOutInt(fromBuffer, msgLength);
         if (msgLength != 0){
            ChanOut(fromBuffer, msg, msgLength);
            free(msg);
         }
      }
}

/**************************************************************************
*
*     This process provides timing data using the high resolution
*     timer (1 µsec) - it acts like a stop watch (on/off/on...) and
*     is run at high priority.
*/
int   stopwatch(procdesc,inc,outc)
int procdesc;
Channel *inc, *outc;
{
      int      start,stop;
      int      toggle;

      while (TRUE){
         toggle = ChanInInt(inc);          /* start timing */
         start = Time();                   /* read start time */
         toggle = ChanInInt(inc);          /* stop timing */
         stop = Time();                    /* read stop time */
         ChanOutInt(outc,stop-start);      /* return elapsed time */
      }
}
/**************************************************************************
*
*     The heart of the workfarm code.  This is where the actual
*     work (computations) is done.
*/
```

```
/* global variables */
int         turnflag;   /* turning point, bottom or surface refl. */
int         done;       /* stop work flag */
int         cnt;
int         bcount;
double      ttime;      /* travel time at one point along ray path */
double      omega;
double      epsilon;
double      scalesave;
double      permScale;

position    *raypath;   /* ray path coordinates */
gauss_beam  *beampath;  /* beam parameters */
complex     *gamma;     /* partial beam results */
int         *branch;    /* sqrt branch for q(s) */
screen_pos  *screen;    /* Mac screen coordinates of ray path */

source_posn    source;         /* source position */
position       receiver[1][1];/* receiver position */
sound_profile  profile;        /* SS profile data */
bottom_profile bprofile;       /* bottom bathymetry data */
work_params    work;           /* various parameters */
beam_params    beamparam;      /* Gauss. beam parameters */
ray_result     finray;         /* end point, etc of ray */
screen_data    s;              /* size, etc of Mac screen */

Channel        *outChan;

int render(procdesc,tolocal, fromlocal, totimer, fromtimer)
int procdesc;
Channel *tolocal, *fromlocal, *totimer, *fromtimer;
{
      int              procID;
      static int       numtimes = 0;
      int              len;
      int              thetime;
      int              temp;
      double           theta;

      outChan = fromlocal;

      /* allocate the memory required to store all the data */
      raypath = (position *)malloc(sizeof(position)*maxraypoints);
      beampath = (gauss_beam *)malloc(sizeof(gauss_beam)*maxraypoints);
      gamma = (complex *)malloc(sizeof(complex)*maxraypoints);
      branch = (int *)malloc(sizeof(int)*maxraypoints);
      screen = (screen_pos *)malloc(sizeof(screen_pos)*maxraypoints);

      while (TRUE) {
         switch(ChanInInt(tolocal)) {
            case MSG_ANGLE:   /* work packet */
               procID = ChanInInt(tolocal);
               len = ChanInInt(tolocal);
               ChanIn(tolocal,&theta,len);
```

```
                    control(theta);
                    break;
                case MSG_SSPROFILE:      /* SS profile data */
                    procID = ChanInInt(tolocal);
                    len = ChanInInt(tolocal);
                    ChanIn(tolocal,&profile,len);
                    break;
                case MSG_BPROFILE:       /* bottom data */
                    procID = ChanInInt(tolocal);
                    len = ChanInInt(tolocal);
                    ChanIn(tolocal,&bprofile,len);
                    break;
                case MSG_BEAM:           /* Gauss. beam params */
                    procID = ChanInInt(tolocal);
                    len = ChanInInt(tolocal);
                    ChanIn(tolocal,&beamparam,len);
                    break;
                case MSG_SOURCE:         /* source position */
                    procID = ChanInInt(tolocal);
                    len = ChanInInt(tolocal);
                    ChanIn(tolocal,&source,len);
                    break;
                case MSG_RECEIVER:       /* receiver position */
                    procID = ChanInInt(tolocal);
                    len = ChanInInt(tolocal);
                    ChanIn(tolocal,&receiver[0][0],len);
                    break;
                case MSG_SCREEN:  /* screen parameters */
                    procID = ChanInInt(tolocal);
                    len = ChanInInt(tolocal);
                    ChanIn(tolocal,&s,len);
                    break;
                case MSG_WORK:    /* misc. parameters */
                    procID = ChanInInt(tolocal);
                    len = ChanInInt(tolocal);
                    ChanIn(tolocal,&work,len);
                    permScale = work.scaleMax;
                    break;
                case MSG_TEST:    /* test message */
                    procID = ChanInInt(tolocal);
                    len = ChanInInt(tolocal);
                    testmsg(procID,len);
                    break;
                default:
                    /* none ! */
                    break;
            }
        }
}
/************************************************************************
*     Send back test message - this is mainly used to send
*     back various 'things'.  It is useful during software
*     development to send back addresses of variables, etc
*/
testmsg(procID,len)
```

```
int      procID,len;
{
      ChanOutInt(outChan,MSG_TEST);
      ChanOutInt(outChan,procID);
      ChanOutInt(outChan,len);
}
/**********************************************************************
*     This function coordinates the various things that need
*     to be done.
*/
void control(theta)
double      theta;      /* theta in degrees */
{
      int         len;
      register    i;

      work.scaleMax = permScale;    /* store scaleMax so we don't */
      scalesave = permScale;        /* lose it */
      finray.bottom_hits  = 0;
      finray.surface_hits = 0;
      RKF4(theta);                  /* do ray trace first */
      GaussSumm();                  /* then the Gauss. beam stuff */

      /* convert raypath to Mac screen coordinates */
      for (i = 0; i < cnt; i++){
         screen[i].h = HRaypos(raypath[i].x);
         screen[i].v = VRaypos(raypath[i].z);
      }

      /* send the data back to the Mac (via a few buffers */
      len = cnt*sizeof(screen_pos);
      ChanOutInt(outChan,MSG_RAY_DATA);
      ChanOutInt(outChan,ourID);
      ChanOutInt(outChan,len);
      ChanOut(outChan,screen,len);
}
/**********************************************************************
*     Converts depth (z) value into vertical pixel location
*     for Mac display
*/
int VRaypos(z)
double z;
{
      return(s.top + s.RayPixelsV*(z/s.RayScaleV));
}

/*
*     Converts range (x) value into horizontal pixel location
*     for Mac display
*/
int HRaypos(r)
double r;
{
      return(s.left + s.RayPixelsH*(r/s.RayScaleH));
}
```

```
/***********************************************************************
*     Computes c, c', c'' for a given depth value using cubic spline
*/
c(z,svel)
double          z;        /* given depth */
sound_speed     *svel;    /* used to store return values */
{
      int       i = 0;
      int       endflag = 0;
      double    w,not_w;
      double    del_z;
      int       step;
      double    endgrad = 0.016;      /* typical depth dependent */
                                      /* gradient */
      /* find upper index for this depth in SSP data */
      i = 0;
      while (profile.z[i] <= z){
         ++i;
         if (i==profile.npts){        /* z value is past the last */
            i -= 1;                   /* tabulated value */
            endflag = 1;
            break;
         }
      }

      /* evaluate speed of sound */
      del_z = profile.z[i] - profile.z[i-1];
      if (endflag == 1)
         w = 1.0;
      else
         w = (z - profile.z[i-1])/del_z;
      not_w = 1.0 - w;

      /* speed of sound */
      svel->c = (not_w)*profile.c[i-1] + w*profile.c[i]
            + del_z*del_z*(profile.a[i]*(pow(w,3.0) - w)
            + profile.a[i-1]*(pow(not_w,3.0) - not_w));
      if (endflag == 1){ /* use linear interpolation */
         del_z = z - profile.z[i];
         svel->c += del_z*endgrad;
         svel->g  = endgrad;
         svel->gg = 0.0;
      }
      else{
         /* speed of sound gradient */
         svel->g = (profile.c[i] - profile.c[i-1])/del_z
               + del_z*(profile.a[i]*(3.0*w*w - 1)
               - profile.a[i-1]*(3.0*not_w*not_w - 1.0));

         /* second derivative of speed of sound */
         svel->gg = 6.0*(not_w*profile.a[i-1] + w*profile.a[i]);
      }
}
/***********************************************************************
*     Compute bottom depth and gradient for a given range value
```

```
*/
bottomval (r,bot)
double          r;        /* given range value */
bottom_point    *bot;     /* used to store return values */
{
      int       i = 0;
      int       endflag = 0;
      double    w,not_w;
      double    del_r;
      int       step;

      /* find upper index for this range in bottom data */
      i = 0;
      while (bprofile.r[i] <= r){
         ++i;
         if (i==bprofile.npts){      /* range is past last */
            i -= 1;                  /* tabulated value */
            endflag = 1;
            break;
         }
      }

      del_r = bprofile.r[i] - bprofile.r[i-1];
      w = (r - bprofile.r[i-1])/del_r;
      not_w = 1.0 - w;

      if(endflag == 1){              /* bottom value is set to last */
         bot->z = bprofile.z[i];     /* tabulated value ie.flat bottom */
         bot->g = 0.0;               /* gradient is zero */
      }
      else{
         bot->z = (not_w)*bprofile.z[i-1] + w*bprofile.z[i]
               + del_r*del_r*(bprofile.a[i]*(pow(w,3.0) - w)
               + bprofile.a[i-1]*(pow(not_w,3.0) - not_w));

         bot->g = -1.0*((bprofile.z[i] - bprofile.z[i-1])/del_r
               + del_r*(bprofile.a[i]*(3.0*w*w - 1)
               - bprofile.a[i-1]*(3.0*not_w*not_w - 1.0)));
      }
}
/***********************************************************************
*     Evaluates ray equation and computes p's q's and travel time
*/
fcn(h,z,b,misc,k)
double          h,z;
gauss_beam      *b;
extra_stuff     *misc;
double          k[5];
{
      sound_speed    svel;
      bottom_point   bot;
      double         temp,c2,cosine;

      /* compute z value */
      if (z < 0.0)      /* this shouldn't happen */
```

```
            c(0.0,&svel);
         else
            c(z,&svel);
         temp = pow(1.0/(misc->a*svel.c),2.0) - 1.0;
         if(temp < 0.0){
            turnflag = turn_pt;  /* turning point !!! */
            k[zval] = -1.0;
         }
         else
            k[zval] = h*sqrt(temp);

         /* compute p's and q's */
         c2 = svel.c*svel.c;
         cosine = cos(misc->angle);
         k[p1val] = h*(-1.0*svel.gg*b->q1/(c2*cosine));
         k[q1val] = h*(svel.c*b->p1/cosine);
         k[p2val] = h*(-1.0*svel.gg*b->q2/(c2*cosine));
         k[q2val] = h*(svel.c*b->p2/cosine);
}

/**********************************************************************
*     Set up initial parameters.
*/
void setup(svel,misc,beam)
sound_speed    *svel;
extra_stuff    *misc;
gauss_beam     *beam;
{
         if(misc->angle == 0.0)        /* a little fudge required */
            misc->angle += 0.001;      /* for transputer version of code */

         misc->angle = misc->angle*PI/180.0; /* convert degrees */
                                             /* to radians */

         misc->a = cos(misc->angle)/svel->c; /* Snell's constant */

         /* determine initial direction of ray */
         if ((misc->angle < 0.0) || (misc->angle == 0.0 && svel->g < 0.0))
            misc->direction = down;
         else
            misc->direction = up;
         misc->angle = fabs(misc->angle);

         /* Gauss. beam I.C.'s */
         omega = 2.0*PI*source.frequency;
         beam->p1 = 1.0;
         beam->q1 = 0.0;
         beam->p2 = 0.0;
         beam->q2 = 1.0;

         /* pick optimum estimate for epsilon */
         switch(beamparam.IC){
            case fillbeams:      /* space filling beams */
               epsilon = 2.0*svel->c*svel->c/(omega*
                         work.angle_incr*work.angle_incr);
            break;
         case minwidthbeams:  /* min. width beams */
            epsilon = svel->c*receiver[0][0].x;
            break;
         case cervenybeams:   /* not implemented yet */
            break;
      }
}
/***********************************************************************
*     Handle turning points and surface reflections
*/
void turnpoint(ray,misc,beam)
position       *ray;
extra_stuff    *misc;
gauss_beam     *beam;
{
      sound_speed    svel, refl_svel;
      double         temp_z,temp_angle,temp_r;
      double         refl_angle,delta_z,delta_r,step,same_g;
      bottom_point   bot;
      int            i;

      switch(turnflag){
         case surf_refl:      /* surface reflection */
            c(ray->z,&svel);
            temp_z = ray->z, temp_angle = misc->angle;
            c(0.0,&refl_svel);
            refl_angle = fabs(acos(cos(temp_angle)*refl_svel.c/svel.c));
            delta_r = fabs((sin(temp_angle) - sin(refl_angle))/
                           (misc->a*svel.g));
            if (ray->x + delta_r < receiver[0][0].x){
               ray->z = 0.0;
               finray.surface_hits += 1;
               tpq(delta_r,misc->angle,&svel,beam);
               increment();
               Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
               ray->x += delta_r;
               misc->direction = -1.0*misc->direction;
               misc->angle = refl_angle;
               raypath[cnt] = *ray;
               beampath[cnt] = *beam;
               sqrtBranch();
            }
            if (ray->x + delta_r < receiver[0][0].x){
               /*ttime += fabs(delta_r/(svel.c*cos(misc->angle)));*/
               tpq(delta_r,misc->angle,&svel,beam);
               increment();
               Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
               ray->z = temp_z, misc->angle = temp_angle;
               ray->x += delta_r;
               raypath[cnt] = *ray;
               beampath[cnt] = *beam;
               sqrtBranch();
            }
            else{
```

```
            c(ray->z,&svel);
            delta_r = receiver[0][0].x - ray->x;
            misc->angle = fabs(asin(sin(misc->angle) +
                          delta_r*misc->a*svel.g));
            delta_z = fabs((cos(misc->angle)/misc->a -
                      svel.c)/svel.g);
            tpq(delta_r,misc->angle,&svel,beam);
            increment();
            Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
            ray->z += misc->direction*delta_z, ray->x += delta_r;
            raypath[cnt] = *ray;
            beampath[cnt] = *beam;
            sqrtBranch();
            done = true;
         }
         break;
      case bott_refl: /* Bottom reflection */
         bottom_step(ray,misc,beam);
         break;
      case turn_pt: /* turning point */
         c(ray->z,&svel);
         same_g = svel.g;  /* use same g throughout for approx. */
         temp_z = ray->z, temp_angle = misc->angle;
         delta_z = fabs(((1.0/misc->a) - svel.c)/same_g);
         delta_r = fabs(sin(misc->angle)/(misc->a*same_g));
         if (delta_r == 0.0){          /* zero start condition */
            ray->z += misc->direction*0.05;
            c(ray->z,&refl_svel);
            misc->angle = fabs(acos(cos(temp_angle)
                          *refl_svel.c/svel.c));
            delta_r = fabs((sin(temp_angle)-sin(misc->angle))/
                      (misc->a*same_g));
            ray->x += delta_r;
            tpq(delta_r,misc->angle,&svel,beam);
            increment();
            Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
            raypath[cnt] = *ray;
            beampath[cnt] = *beam;
            sqrtBranch();
         }
         else{
            if ((ray->x + delta_r) < receiver[0][0].x){
               ray->z += misc->direction*delta_z;
               ray->x += delta_r;
               bottomval(ray->x,&bot);
               if (ray->z > bot.z){
                  ray->z -= misc->direction*delta_z;
                  ray->x -= delta_r;
                  bottom_step(ray,misc,beam);
                  break;
               }
               else{
                  tpq(delta_r,misc->angle,&svel,beam);
                  increment();
                  Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
                  misc->angle = 0.0;
                  misc->direction = -1.0*misc->direction;
                  raypath[cnt] = *ray;
                  beampath[cnt] = *beam;
                  sqrtBranch();
               }
            }
            if ((ray->x + delta_r) < receiver[0][0].x){
               bottomval(ray->x + delta_r,&bot);
               if (temp_z > bot.z){
                  bottom_step(ray,misc,beam);
                  break;
               }
               else{
                  ray->z = temp_z;
                  ray->x += delta_r;
                  tpq(delta_r,misc->angle,&svel,beam);
                  increment();
                  Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
                  misc->angle = temp_angle;
                  raypath[cnt] = *ray;
                  beampath[cnt] = *beam;
                  sqrtBranch();
               }
            }
            else{
               c(ray->z,&svel);
               delta_r = receiver[0][0].x - ray->x;
               misc->angle = fabs(asin(sin(misc->angle) +
                                 delta_r*misc->a*same_g));
               delta_z = fabs(((cos(misc->angle)/misc->a) -
                          svel.c)/same_g);
               tpq(delta_r,misc->angle,&svel,beam);
               increment();
               Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
               ray->z += misc->direction*delta_z, ray->x += delta_r;
               raypath[cnt] = *ray;
               beampath[cnt] = *beam;
               sqrtBranch();
               done = true;
            }
         }
         break;
      } /* end of switch */
}

/**********************************************************************
*     Steps ray path to, and out from bottom.
*/
bottom_step(ray,misc,beam)
position    *ray;
extra_stuff *misc;
gauss_beam  *beam;
{
      int            i;
```

```
sound_speed     svel,refl_svel;
bottom_point    bot;
double          step;
double          delta_z,delta_r,refl_angle;

c(ray->z,&svel);
step = 0.0,delta_z = 0.0,delta_r = 0.0;
refl_angle = misc->angle;
bottomval(ray->x,&bot);
step = 0.5*(bot.z - ray->z);
i = 0;
while (fabs(bot.z-(ray->z+misc->direction*delta_z)) >
      work.bottom_tolerance){
   ++i;
   c(ray->z+misc->direction*(delta_z+step),&refl_svel);
   refl_angle = fabs(acos(cos(misc->angle)*refl_svel.c/svel.c));
   delta_r = fabs((sin(misc->angle) - sin(refl_angle))/
             (misc->a*svel.g));
   bottomval(ray->x + delta_r,&bot);
   if ((bot.z > (ray->z+misc->direction*(delta_z+step))))
      delta_z += step;
   else
      step = 0.5*step;
}
/* step ray into bottom */
ray->z += misc->direction*delta_z, ray->x += delta_r;
tpq(delta_r,misc->angle,&svel,beam);
increment();
Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
misc->angle = refl_angle;
c(ray->z,&refl_svel);   /* compute svel at bottom */
if ((misc->direction = down) && ((refl_angle +
     2.0*atan(bot.g)) > 0.0))
   misc->direction = -1.0*misc->direction;
finray.bottom_hits += 1;
raypath[cnt] = *ray;
beampath[cnt] = *beam;
sqrtBranch();

/* step ray to next point */
misc->angle += 2.0*atan(bot.g);      /* new angle after */
                                     /* bottom reflection */
misc->angle = fabs(misc->angle);
if (misc->angle > PI/2.0)
   done = true;

/* compute new 'a' value */
misc->a = cos(misc->angle)/refl_svel c;
delta_z = 10.0;
c(ray->z,&svel);
c(ray->z+misc->direction*delta_z,&refl_svel);
refl_angle = fabs(acos(cos(misc->angle)*refl_svel.c/svel.c));
delta_r = fabs((sin(misc->angle) - sin(refl_angle))/
          (misc->a*svel.g));
      ray->z += misc->direction*delta_z;
      ray->x += delta_r;
      tpq(delta_r,misc->angle,&svel,beam);
      increment();
      Gamm(-1.0*misc->angle*misc->direction,&svel,beam);
      misc->angle = refl_angle;
      raypath[cnt] = *ray;
      beampath[cnt] = *beam;
      sqrtBranch();
}
/***********************************************************************
*     Compute travel time and p's and q's (tpq's) when using linear
*     approximation method (ie. at turning points, etc)
*/
tpq(delta_r,angle,svel,beam)
double      delta_r,angle;
sound_speed *svel;
gauss_beam  *beam;
{
      double   temp_p,cosine,c2;

      cosine = cos(angle);
      c2 = svel->c*svel->c;
      ttime += delta_r/(svel->c*cosine);  /* compute new time */

      /* compute new p's & q's */
      temp_p = beam->p1;
      beam->p1 += delta_r*(-1.0*svel->gg*beam->q1/(c2*cosine));
      beam->q1 += delta_r*(svel->c*temp_p/cosine);
      temp_p = beam->p2;
      beam->p2 += delta_r*(-1.0*svel->gg*beam->q2/(c2*cosine));
      beam->q2 += delta_r*(svel->c*temp_p/cosine);
}

/***********************************************************************
*     The Runge-Kutta-Fehlberg algorithm
*/
/* Vars Global to RKF and reducestep routines */
double         hstart = 25.0; /* starting step size */
double         h;    /* step size */
position       ray;  /* single ray path position */
extra_stuff    misc;
gauss_beam     gb;

void RKF4(theta)
double         theta;
{

      double         k[7][5];
      bottom_point   bot;
      sound_speed    svel;
      gauss_beam     newb;
      double         newz;
      double         scale;
      double         err,zstep;
```

```
misc.angle = theta;
ray.z = source.z, ray.y = source.y, ray.x = source.x;
c(ray.z,&svel);
setup(&svel,&misc,&gb);
cnt = 0;
bcount = 0;
h = hstart;
ttime = 0.0;
done = false;
raypath[cnt] = ray;
beampath[cnt] = gb;
Gamm(-1.0*misc.angle*misc.direction,&svel,&gb);

while(!done){
   newz = ray.z;
   newb = gb;
   fcn(h,newz,&newb,&misc,&k[1]); /* first fcn evaluation */
   if (k[1][zval] < 0.0){
      reducestep();
      continue;
   }
   newz = ray.z + misc.direction*k[1][zval]/4.0;
   newb.p1 = gb.p1 + k[1][p1val]/4.0;
   newb.p2 = gb.p2 + k[1][p2val]/4.0;
   newb.q1 = gb.q1 + k[1][q1val]/4.0;
   newb.q2 = gb.q2 + k[1][q2val]/4.0;
   fcn(h,newz,&newb,&misc,&k[2]); /* second fcn evaluation */
   if (k[2][zval] < 0.0){
      reducestep();
      continue;
   }
   newz = ray.z+misc.direction*
           (3.0*k[1][zval]+9.0*k[2][zval])/32.0;
   newb.p1 = gb.p1 + (3.0*k[1][p1val]+9.0*k[2][p1val])/32.0;
   newb.p2 = gb.p2 + (3.0*k[1][p2val]+9.0*k[2][p2val])/32.0;
   newb.q1 = gb.q1 + (3.0*k[1][q1val]+9.0*k[2][q1val])/32.0;
   newb.q2 = gb.q2 + (3.0*k[1][q2val]+9.0*k[2][q2val])/32.0;
   fcn(h,newz,&newb,&misc,&k[3]); /* third fcn evaluation */
   if (k[3][zval] < 0.0){
      reducestep();
      continue;
   }
   newz = ray.z+misc.direction*(1932.0*k[1][zval]-
           7200.0*k[2][zval]+
           7296.0*k[3][zval])/2197.0;
   newb.p1 = gb.p1+(1932.0*k[1][p1val]-7200.0*k[2][p1val]+
                7296.0*k[3][p1val])/2197.0;
   newb.p2 = gb.p2+(1932.0*k[1][p2val]-7200.0*k[2][p2val]+
                7296.0*k[3][p2val])/2197.0;
   newb.q1 = gb.q1+(1932.0*k[1][q1val]-7200.0*k[2][q1val]+
                7296.0*k[3][q1val])/2197.0;
   newb.q2 = gb.q2+(1932.0*k[1][q2val]-7200.0*k[2][q2val]+
                7296.0*k[3][q2val])/2197.0,
   fcn(h,newz,&newb,&misc,&k[4]); /* fourth fcn evaluation */
```

```
if (k[4][zval] < 0.0){
   reducestep();
   continue;
}
newz = ray.z + misc.direction*(439.0*k[1][zval]/216.0 -
       8.0*k[2][zval] + 3680.0*k[3][zval]/513.0 -
       845.0*k[4][zval]/4104.0);
newb.p1 = gb.p1 + (439.0*k[1][p1val]/216.0 -
          8.0*k[2][p1val] + 3680.0*k[3][p1val]/513.0 -
          845.0*k[4][p1val]/4104.0);
newb.p2 = gb.p2 + (439.0*k[1][p2val]/216.0 -
          8.0*k[2][p2val] + 3680.0*k[3][p2val]/513.0 -
          845.0*k[4][p2val]/4104.0);
newb.q1 = gb.q1 + (439.0*k[1][q1val]/216.0 -
          8.0*k[2][q1val] + 3680.0*k[3][q1val]/513.0 -
          845.0*k[4][q1val]/4104.0);
newb.q2 = gb.q2 + (439.0*k[1][q2val]/216.0 -
          8.0*k[2][q2val] + 3680.0*k[3][q2val]/513.0 -
          845.0*k[4][q2val]/4104.0);
fcn(h,newz,&newb,&misc,&k[5]); /* fifth fcn evaluation */
if (k[5][zval] < 0.0){
   reducestep();
   continue;
}
newz = ray.z + misc.direction*(-8.0*k[1][zval]/27.0 +
       2.0*k[2][zval] - 3544.0*k[3][zval]/2565.0 +
       1859.0*k[4][zval]/4104.0 - 11.0*k[5][zval]/40.0);
newb.p1 = gb.p1 + (-8.0*k[1][p1val]/27.0 +
          2.0*k[2][p1val] - 3544.0*k[3][p1val]/2565.0 +
          1859.0*k[4][p1val]/4104.0 - 11.0*k[5][p1val]/40.0);
newb.p2 = gb.p2 + (-8.0*k[1][p2val]/27.0 +
          2.0*k[2][p2val] - 3544.0*k[3][p2val]/2565.0 +
          1859.0*k[4][p2val]/4104.0 - 11.0*k[5][p2val]/40.0);
newb.q1 = gb.q1 + (-8.0*k[1][q1val]/27.0 +
          2.0*k[2][q1val] - 3544.0*k[3][q1val]/2565.0 +
          1859.0*k[4][q1val]/4104.0 - 11.0*k[5][q1val]/40.0);
newb.q2 = gb.q2 + (-8.0*k[1][q2val]/27.0 +
          2.0*k[2][q2val] - 3544.0*k[3][q2val]/2565.0 +
          1859.0*k[4][q2val]/4104.0 - 11.0*k[5][q2val]/40.0);
fcn(h,newz,&newb,&misc,&k[6]); /* final fcn evaluation */
if (k[6][zval] < 0.0){
   reducestep();
   continue;
}
/* compute error estimate */
err = fabs(k[1][zval]/360.0 - 128.0*k[3][zval]/4275.0 -
      2197.0*k[4][zval]/75240.0 +
      k[5][zval]/50.0 + 2.0*k[6][zval]/55.0);

if (err/h < work.tolerance){     /* error is acceptable */
   /* compute new values of z, p and q */
   zstep = misc.direction*(25.0*k[1][zval]/216.0 +
           1408.0*k[3][zval]/2565.0 +
           2197.0*k[4][zval]/4104.0 - k[5][zval]/5.0);
   gb.p1 += (25.0*k[1][p1val]/216.0 +
                1408.0*k[3][p1val]/2565.0 +
                2197.0*k[4][p1val]/4104.0 - k[5][p1val]/5.0);
         gb.p2 += (25.0*k[1][p2val]/216.0 +
                1408.0*k[3][p2val]/2565.0 +
                2197.0*k[4][p2val]/4104.0 - k[5][p2val]/5.0);
         gb.q1 += (25.0*k[1][q1val]/216.0 +
                1408.0*k[3][q1val]/2565.0 +
                2197.0*k[4][q1val]/4104.0 - k[5][q1val]/5.0);
         gb.q2 += (25.0*k[1][q2val]/216.0 +
                1408.0*k[3][q2val]/2565.0 +
                2197.0*k[4][q2val]/4104.0 - k[5][q2val]/5.0);
         ray.z += zstep;
         ray.x += h;
         if (ray.z < 0.0){  /* check if we surfaced */
            turnflag = surf_refl;
            ray.z -= zstep; /* step back */
            ray.x -= h;
            reducestep();   /* reduce step size */
            continue;
         }
         bottomval(ray.x,&bot); /* check if we bottomed out */
         if ((ray.z > bot.z) && (fabs(ray.z - bot.z) >
            work.bottom_tolerance)){
            turnflag = bott_refl;
            ray.z -= zstep; /* step back */
            ray.x -= h;
            reducestep();   /* reduce step size */
            if (misc.angle > PI/2.0)
               done = true;
            continue;
         }
         ttime += h/(svel.c*cos(misc.angle));
         if (ray.x >= receiver[0][0].x) /* past receiver range */
            done = true;
         increment();
         Gamm(-1.0*misc.angle*misc.direction,&svel,&gb);
         raypath[cnt] = ray;
         beampath[cnt] = gb;
         sqrtBranch();
         c(ray.z,&svel);
         misc.angle = acos(svel.c*misc.a);
      }

      /* scale step size */
      scale = 0.84*pow((work.tolerance*h/err),0.25);
      if (scale < work.scaleMin) scale = work.scaleMin;
      if (scale > work.scaleMax) scale = work.scaleMax;
      h = h*scale;
      if(h < 1.0){   /* step is kind of small !?!*/
         reducestep();
         continue;
      }
      if ((ray.x + h) > receiver[0][0].x) /* don't step past */
                                          /* receiver */
         h = receiver[0][0].x - ray.x;
```

```
        }
        finray.rayend = ray;    /* return final ray position */
        finray.angle = misc.angle;
        finray.time = ttime;
}
/*************************************************************************
*       Reduces integration step size
*/
reducestep()
{
        work.scaleMax = 1.0;
        h = h/2.0;
        if (h < hstart){
           work.scaleMax = scalesave; /* restore scalemax to */
                                      /* original value */
           turnpoint(&ray,&misc,&gb); /* go to turning point approx. */
           h = hstart;
        }
}
/*************************************************************************
*       Increments array index used for storing ray path.  Also
*       checks if we have too many points
*/
increment()
{
        cnt++;
        if(cnt >= maxraypoints)
           done = true;
}
/*************************************************************************
*       Checks for p(s) imag. axis crossings.
*
sqrtBranch()
{
        double      q1,q2;
        if(cnt == 0)
           branch[0] = 0;
        else{
           q1 = beampath[cnt-1].q2, q2 = beampath[cnt].q2;
           if(((q1 < 0.0)&&(q2 > 0.0)) || ((q1 > 0.0)&&(q2 < 0.0))){
              branch[bcount] = cnt; /* store index where crossing */
                                    /* occurred */
              bcount++;
           }
        }
}
/*************************************************************************
*
*       Complex math functions
*/
complex comp(x,y)                       /* assign reals to type complex */
double      x,y;
{
        complex  z;
        z.re = x;
        z.im = y;
        return(z);
}

complex cadd(x,y)                       /* complex add */
complex     x,y;
{
        complex  z;

        z.re = x.re + y.re;
        z.im - x.im + y.im;
        return(z);
}

complex  cmult(x,y)                     /* complex multiply */
complex     x,y;
{
        complex  z;

        z.re = x.re*y.re - x.im*y.im;
        z.im = x.re*y.im + x.im*y.re;
        return(z);
}

complex cmultd(x,y)                     /* complex times a constant */
double      x;
complex     y;
{
        complex  z;

        z.re = y.re*x;
        z.im = y.im*x;
        return(z);
}

complex cdiv(x,y)                       /* complex division */
complex     x,y;
{
        complex  z;
        double   den;

        if (y.re == 0.0 && y.im == 0.0){     /* divide by zero */
           z.re = HUGE_VAL, z.im = HUGE_VAL;
           return(z);
        }
        else{
           den = y.re*y.re + y.im*y.im;
           z.re = (x.re*y.re + x.im*y.im)/den;
           z.im = (y.re*x.im - x.re*y.im)/den;
           return(z);
        }
}
```

```
double real(x)                  /* return real part */
complex     x;
{
      return(x.re);
}

double imag(x)                  /* return complex part */
complex     x;
{
      return(x.im);
}
/**********************************************************************
*     Computes part of Gauss. beam equation at same time
*     as ray path is computed.
*/
Gamm(theta,svel,beam)
double      theta;
sound_speed *svel;
gauss_beam  *beam;
{
      double   c2;
      double   cs,cn,tr,tz;
      complex  p,q,res,comp(),cdiv(),cmultd();

      tr = cos(theta);
      tz = sin(theta);
      c2 = svel->c*svel->c;
      cs = svel->g*tz;
      cn = -1.0*svel->g*tr;
      p = comp(beam->p1,beam->p2*epsilon);
      q = comp(beam->q1,beam->q2*epsilon);
      res = cdiv(p,q);
      gamma[cnt].re = 0.5*res.re*tr*tr + 2.0*cn*tz*tr/c2 - cs*tz*tz/c2;
      gamma[cnt].im = 0.5*res.im*tr*tr;
}
/**********************************************************************
*     Compute Gauss. beam summation at particular receiver location
*/
GaussSumm()
{
      register i;
      double      temp;
      complex     p,q,comp(),cdiv();

      for (i = 0; i < cnt; i++){
         beampath[i].p2 = beampath[i].p2*epsilon;
         beampath[i].q2 = beampath[i].q2*epsilon;
      }
}
/**********************************************************************
*     Route messages back to Mac
*/
int feedback(procdesc,fromlocal, fromnext, toprev)
int procdesc;
Channel *fromlocal, *fromnext, *toprev;
```

```
{
      char          *buff;
      int           msg,idx,len;
      int           procID,atime;
      Channel       *chan;
      static int    toggle = 0;

      ChanOutInt(toprev,MSG_BOOTED); /* tell host we're alive */

      while (TRUE) {

         /* NOTE: the code below alternates between two calls to      */
         /* the same function ProcAlt.  The ProcAlt function tends    */
         /* to favor the channel given first in the argument list     */
         /* if both are ready at the same time. The two calls simply  */
         /* alternates the order of the arguments for a more          */
         /* equitable approach when both channels are ready for input */

         switch(toggle){
            case 0:
               do {
                  idx = ProcAlt(fromlocal,fromnext,0);
               } while (idx==-1);
               switch(idx) {
                  case 0: chan = fromlocal;     break;
                  case 1: chan = fromnext;      break;
               }
               toggle = !toggle;
               break;
            case 1:
               do {
                  idx = ProcAlt(fromnext,fromlocal,0);
               } while (idx==-1);
               switch(idx) {
                  case 0: chan = fromnext;      break;
                  case 1: chan = fromlocal;     break,
               }
               toggle = !toggle;
               break;
         }
         msg = ChanInInt(chan);
         switch(msg) {
            case MSG_BOOTED:
               ChanOutInt(toprev,MSG_BOOTED);
               break;
            case MSG_TEST:
               procID = ChanInInt(chan);
               len = ChanInInt(chan);
               ChanOutInt(toprev,MSG_TEST);
               ChanOutInt(toprev,procID);
               ChanOutInt(toprev,len);
               break;
            case MSG_WORK_DONE:
               procID = ChanInInt(chan);
               len = ChanInInt(chan);
```

```
                buff = (char *)malloc(len);
                ChanIn(chan,buff,len);
                ChanOutInt(toprev,MSG_WORK_DONE);
                ChanOutInt(toprev,procID);
                ChanOutInt(toprev,len);
                ChanOut(toprev,buff,len);
                free(buff);
                break;
            case MSG_RAY_DATA:
                procID = ChanInInt(chan);
                len = ChanInInt(chan);
                buff = (char *)malloc(len);
                ChanIn(chan,buff,len);
                ChanOutInt(toprev,MSG_RAY_DATA);
                ChanOutInt(toprev,procID);
                ChanOutInt(toprev,len);
                ChanOut(toprev,buff,len);
                free(buff);
                break;
        }
    }
}

/*************************************************************************
*     This is the MAIN part of the program (MAIN in the C context).
*     This is where all the process allocation and prioritization and
*     channel allocation is handled.
*/

/* Declare channels */
Channel *tolocal, *fromlocal; /* non-link channels */
Channel *toInBuffer0,*toInBuffer1,*toInBuffer2;
Channel *fromOutBuffer0,*fromOutBuffer1,*fromOutBuffer2;
Channel *totimer, *fromtimer;

/* Declare processes */
Process *pthroughput, *prender, *pfeedback;
Process *pInBuffer0,*pInBuffer1,*pInBuffer2;
Process *pOutBuffer0,*pOutBuffer1,*pOutBuffer2;
Process *pstopwatch;

extern char *_heapend;          /* points to end of heap space */
extern char *_heapstart;        /* points to start of heap space */

main() {

    char  chipRAM[CHIPRAMSIZE]; /* allocate memory from on-chip RAM */

    _heapend = 0x80101000;          /* 1 MByte Module */

    /* allocate the internal channels */
    tolocal         = ChanAlloc();
    fromlocal       = ChanAlloc();
    toInBuffer0     = ChanAlloc();
    toInBuffer1     = ChanAlloc();
```

```
toInBuffer2     = ChanAlloc();
fromOutBuffer0 = ChanAlloc();
fromOutBuffer1 = ChanAlloc();
fromOutBuffer2 = ChanAlloc();
totimer         = ChanAlloc();
fromtimer       = ChanAlloc();

prender = (Process *)malloc(sizeof(Process));
ProcInit(prender,render,chipRAM,CHIPRAMSIZE,4,tolocal,fromlocal,to
         timer,fromtimer);

/* declare throughput process */
pthroughput = ProcAlloc(throughput,STACKSIZE,3,
              LINK0IN,LINK1OUT,toInBuffer2);

/* declare input buffers */
pInBuffer2  = ProcAlloc(buffer,100,2,toInBuffer2,toInBuffer1);
pInBuffer1  = ProcAlloc(buffer,100,2,toInBuffer1,toInBuffer0);
pInBuffer0  = ProcAlloc(buffer,100,2,toInBuffer0,tolocal);

/* declare output buffers */
pOutBuffer0 = ProcAlloc(buffer,100,2,fromlocal,fromOutBuffer0);
pOutBuffer1 = ProcAlloc(buffer,100,2,fromOutBuffer0,
                        fromOutBuffer1);
pOutBuffer2 = ProcAlloc(buffer,100,2,fromOutBuffer1,
                        fromOutBuffer2);

/* declare feedback process */
pfeedback   = ProcAlloc(feedback,STACKSIZE,3,fromOutBuffer2,
                        LINK1IN,LINK0OUT);
/* declare stopwatch process */
pstopwatch  = ProcAlloc(stopwatch,1000,2,totimer,fromtimer);

/* launch the processes in parallel */
ProcRun(prender); /* low priority */
ProcToHigh();     /* switch to high priority */
ProcPar(pthroughput,pfeedback,pstopwatch,pInBuffer0,
        pInBuffer1,pInBuffer2,pOutBuffer0,
         pOutBuffer1,pOutBuffer2,0);
}
```

D. INCLUDE FILES

The following are two of the include (.h) files used for the ray tracing algorithm. The first, *messageID.h*, gives the message identification number definitions. The second, *raydefs.h*, gives the definitions for some of the constants and data structures used throughout the ray tracing and associated algorithms.

1. messageID.h

```
/*
 *          Message ID's and definitions
 */

/* Messages from Mac to transputer */
#define  MSG_PASS        0     /* pass data to next processor      */
#define  MSG_PROCID      1     /* assign processor ID number       */
#define  MSG_SSPROFILE   2     /* send sound velocity profile data */
#define  MSG_BPROFILE    3     /* send sound velocity profile data */
#define  MSG_WORK        4     /* send work parameters             */
#define  MSG_BEAM        5     /* send beam parameters             */
#define  MSG_ANGLE       6     /* send work packet                 */
#define  MSG_SOURCE      7     /* send source position             */
#define  MSG_RECEIVER    8     /* send receiver position           */
#define  MSG_TEST        9     /* used for testing                 */
#define  MSG_SCREEN      10    /* used send Mac screen parameters  */

/* Messages from transputer to Mac */
#define  MSG_RAY_DATA    21    /* return ray data                  */
#define  MSG_BEAM_DATA   22    /* return beam data                 */
#define  MSG_TIME_DATA   23    /* return timing info               */
#define  MSG_ERROR       999   /* not implemented                  */
#define  MSG_BOOTED      555   /* tell host we booted OK           */
```

2. raydefs.h

```
#define  maxpoints 50        /* max. number of points for SS profile */
#define  maxbottompoints 100  /* max. number of points for bottom */
#define  maxraypoints 2000 /* max. number of points for ray path, etc */

#define  nil      0L

#define  up    -1.0
#define  down   1.0

#define  surf_refl   1  /* surface reflection */
#define  bott_refl   2  /* bottom reflection  */
```

```
#define  turn_pt     3  /* turning point      */

#define  zval  0
#define  p1val 1
#define  p2val 2
#define  q1val 3
#define  q2val 4

/* beam parameter definitions */
#define  incoherent     0
#define  coherent       1
#define  semicoherent   2
#define  fillbeams      0
#define  cervenybeams   1
#define  minwidthbeams  2

#define PI 3.14159265

typedef struct{
      double   angle;
      double   a;
      double   direction;
} extra_stuff;

typedef struct{                /* sound speed profile data structure */
      int      npts;           /* number of tabulated points */
      double   range;          /* range (for range dependency) */
      double   a[maxpoints];   /* spline coefficients */
      double   z[maxpoints];   /* depth values */
      double   c[maxpoints];   /* speed of sound values */
} sound_profile;

typedef struct{                /* bottom bathymetry data structure */
      int      npts;           /* number of tabulated points */
      double   a[maxbottompoints]; /* spline coefficients */
      double   z[maxbottompoints]; /* depth values */
      double   r[maxbottompoints]; /* range values */
} bottom_profile;

typedef struct{       /* bottom point data structure */
      double   z;     /* depth of bottom */
      double   g;     /* gradient of bottom */
} bottom_point;

typedef struct{       /* sound speed data structure */
      double   c;     /* sound speed */
      double   g;     /* sound speed gradient */
      double   gg;    /* second derivative of sound speed */
} sound_speed;

typedef struct{       /* position data structure */
      double   x;     /* range value */
      double   y;     /* azimuthal value (not used) */
      double   z;     /* depth value */
} position;
```

```
typedef struct{                           /* source data structure */
      double   x;
      double   y;
      double   z;
      double   frequency;                 /* source frequency */
} source_posn;

typedef struct{                           /* work parameters data structure */
      double   theta_upper;               /* upper ray angle */
      double   theta_lower;               /* lower ray angle */
      double   angle_incr;                /* angle increment */
      int      numRays;                   /* number of rays to trace */
      double   scaleMax;                  /* max step size scale value */
      double   scaleMin;                  /* min step size scale value */
      double   tolerance;                 /* integration error tolerance */
      double   bottom_tolerance;          /* bottom tolerance value */
} work_params;

typedef struct{             /* Gauss. beam parameters data st1 .cture */
      int      radii;       /* max. beam radii for windowing */
      int      summ;        /* beam summation method */
      int      IC;          /* beam initial condition type */
} beam_params;

typedef struct{                           /* ray results data structure */
      position    rayend;                 /* final position of ray path */
      double      angle;                  /* final angle */
      double      time;                   /* travel time */
      int         surface_hits;           /* number of surface reflections */
      int         bottom_hits;            /* number of bottom reflections */
} ray_result;

typedef struct{                     /* ray path data structure */
      position ray;                 /* ray position */
      double      angle;            /* ray angle */
      double      time;             /* time of travel */
} ray_path;

typedef struct{                     /* Gauss. beam data structure */
      double   p1;                  /* real(p) */
      double   p2;                  /* imag(p) */
      double   q1;                  /* real(p) */
      double   q2;                  /* imag(q) */
} gauss_beam;

typedef struct{                     /* complex number data structure */
      double   re;                  /* real part */
      double   im;                  /* imaginary part */
} complex;

typedef struct{                     /* exponential form of complex # */
      double   r;                   /* magnitude */
      double   theta;               /* phase */
}complexExp;
```

```
typedef struct{                 /* screen coordinate data structure */
      int      h;               /* horizontal position */
      int      v;               /* vertical position */
} screen_pos;

typedef struct{                 /* screen parameters data structure */
      int      top;             /* top of drawing area */
      int      left;            /* left side of drawing area */
      int      RayPixelsV;      /* number of vertical pixels */
      int      RayPixelsH;      /* number of horizontal pixels */
      int      RayScaleV;       /* vertical scale */
      int      RayScaleH;       /* horizontal scale */
} screen_data;

typedef struct{        /* timing information data structure */
      int      t1;     /* ray path calculation time */
      int      t2;     /* screen coord. computation time */
      int      t3;     /* send data to buffer process time */
} tinfo;
```

APPENDIX B

MACINTOSH APPLICATION STRUCTURE

This appendix gives the basic structure of a typical Macintosh application main event loop. For clarity, this example is written using mostly English statements rather than C code.

```
main()
{
      Initialize necessary toolbox routines

      do {

         perform a system task
         check if any data ready from transputers (used for this project
                                                     specifically)
         check to see if an 'event' has occurred
         if so, handle it depending on the type of event
            switch (event type)
            {
               case mouse button was pressed
                  find out where it was pressed
                  handle it depending on where it was pressed
                  switch (where pressed)
                  {
                     case in a Desk Accessory
                        let DA handle it
                        break;
                     case in the menu bar
                        handle the command selected
                        break;
                     case in the drag region of a window
                        drag the window
                        break;
                     case in the content region of a window
                        do appropriate action for content selected
                        break;
                     case in the window's close box
                        track the mouse and close window if necessary
                        break;
                     case in the grow region of the window
                        change size of window appropriately
                        break;
                     case in zoom (in) box of window
                        track mouse and zoom window if necessary
                        break;
                        case in zoom (out) box of window
                           track mouse and zoom window if necessary
                           break;
                     }
                     break;
                  case a key was pressed
                     perform appropriate action for key(s) pressed
                     break;
                  case a window has been activated
                     activate new window and deactivate old window
                     break;
                  case a window requires updating
                     update contents of window
                     break;
            }

      } while (!doneFlag); /* while user has not selected quit */
      return(0);
}
```

LIST OF REFERENCES

Apple Computer, Inc., *Inside Macintosh,* Vols. I-V, Addison-Wesley, 1988.

Brekhovskikh, L., *Waves in a Layered Media,* 2nd ed., Academic Press, 1980.

Brekhovskikh, L., and Lysanov, Y., *Fundamentals of Ocean Acoustics,* Springer-Verlag, 1982.

Bryant, Gregory R., Design, *Implementation and Evaluation of an Abstract Programming and Communications Interface for a Network of Transputers,* MS Thesis, Naval Postgraduate School, Monterey, CA, June 1988.

Červený, V., and Popov, M.M., and Pšenčík, I., "Computation of Wave Fields in Inhomogeneous Media - Gaussian Beam Approach", *Geophys. J. R. Astron. Soc.*, 70, pp. 109-128, 1982.

Clay, C.S., and Medwin H., *Acoustical Oceanography: Principles and Applications,* John Wiley & Sons, 1977.

Davidson, C., "Transputers and How to Feed Them", *APDALog,* pp. 5-7, January 1988.

Gerald, C.F., and Wheatley, P.O., *Applied Numerical Analysis,* 3rd ed., Addison-Wesley Publishing Company, 1985.

Howe, C.D., and Moxon, B., "How to Program Parallel Processors", *IEEE Spectrum,* pp. 36-41, September 1987.

INMOS Corporation, *An Introduction To Transputers,* Draft 2.0, 12 January 1988.

INMOS Corporation, *IMS T800 Transputer*, Engineering Data Booklet, April 1987.

INMOS Limited, *occam® 2 Reference Manual*, Prentice-Hall, 1988.

INMOS Corporation, Technical Note 6, *IMS T800 Architecture*, 1986.

INMOS Corporation, Technical Note 7, *Exploiting Concurrency; A Ray Tracing Example*, by J. Packer, no date given.

INMOS Corporation, Technical Note 17, *Performance Maximization*, by P. Atkin, March 1987.

INMOS Corporation, Technical Note 26, *Notes on Graphics Support and Performance Improvements on the IMS T800*, by G. Harriman, 29 June 1987.

INMOS Corporation, Technical Note 27, *Lies, Damned Lies and Benchmarks*, by R. Shepard and P. Thompson, no date given.

INMOS Corporation, *The Transputer Family*, Product Overview, June 1986.

Kinsler, L.E., and others, *Fundamentals of Acoustics*, 3rd ed., John Wiley & Sons, 1982.

Madariaga, R., "Gaussian Beam Synthetic Seismograms in a Vertically Varying Medium", *Geophys. J. R. Astron. Soc.*, 79, pp. 589-612, 1984.

Maron, M.J., *Numerical Analysis: A Practical Approach*, MacMillan Publishing Co., Inc., 1982.

Naval Postgraduate School Technical Report NPS62-90-003, *The Viability of Acoustic Tomography in Monitoring the Circulation of Monterey Bay*, by J.H. Miller, L.L. Ehret, R.C. Dees, and T.M. Rowan, December 1989.

Moler, C.B., and Solomon L.P., "Use of Splines and Numerical Integration in Geometrical Acoustirs", *J. Acoust. Soc. Am.*, 48 (3), pp. 739-744, 1970.

Munk, W. and Wunsch, C., "Ocean Acoustic Tomography: A Scheme for Large Scale Monitoring", *Deep-Sea Research*, Vol. 26A, pp. 123-161, 1979.

Pedersen, M.A., "Acoustic Intensity Anomolies Introduced by Constant Velocity Gradients", *J. Acoust. Soc. Am.*, 33, pp. 465-474, 1961.

Porter, M.B., and Bucker, H.P., "Applications of Gaussian Beam Tracing to Two- and Three-Dimensional Problems in Ocean Acoustics", paper presented at Paris-IMACS meeting, Copy received 10 October 1989.

Porter, M.B., and Bucker, H.P., "Gaussian Beam Tracing for Computing Ocean Acoustic Fields", *J. Acoust. Soc. Am.*, 82 (4), pp. 1349-1359, 1987.

Spindel, R.C., "Ocean Acoustic Tomography: A Review", *Current Practices and New Technology in Ocean Engineering*, Vol. 11, pp. 7-13, 1986.

Stone, H.S., *High-Performance Computer Architecture*, Addison-Wesley Publishing Company, 1987.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Main Library National Defence Headquarters MGen George R. Pearkes Bldg Ottawa, Ontario Canada K1A 0K2	2
4.	Dr. James H. Miller, Code EC/Mr Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5002	10
5.	Dr. Ching-Sang Chiu, Code OC/Ci Department of Oceanography Naval Postgraduate School Monterey, California 93943-5002	2
6.	Dr. Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, California 93943-5002	2

7. Chairman, Code EC 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5002

8. Capt Roderick S. Scott, Canadian Forces 2
15 Rigel Rd.
Ottawa, Ontario
Canada, K1K 0A1

9. Dr. Lawrence L. Ziomek, Code EC/Zi 1
Department of Electrical and Computer Engineering
Naval Postgraduate School
Monterey, California 93943-5002

10. National Defence Headquarters 1
MGen George R. Pearkes Bldg
Ottawa, Ontario Canada
K1A 0K2
Attention: DMAEM 4-2

11. Lt(N) Don Smith, Canadian Forces 1
Weapons Curricular Office, UX91
Naval Postgraduate School
Monterey, California 93943-5002

12. Lt(N) Ed Chaulk, Canadian Forces 1
Weapons Curricular Office, WT91
Naval Postgraduate School
Monterey, California 93943-5002